January 1992

MTR11229

M. J. Prelle
A. M. Wollrath

The SaM Synchronization Manager Distributed Object-Oriented Programming FY91 Final Report

MITRE
Bedford, Massachusetts

January 1992

MTR11229

M. J. Prelle
A. M. Wollrath

The SaM Synchronization Manager Distributed Object-Oriented Programming FY91 Final Report

CONTRACT SPONSOR RL/C3AB
CONTRACT NO. F19628-89-C-0001
PROJECT NO. 7850
DEPT. G110

MITRE

The MITRE Corporation
Bedford, Massachusetts

Department Approval: Myra Jean Prelle

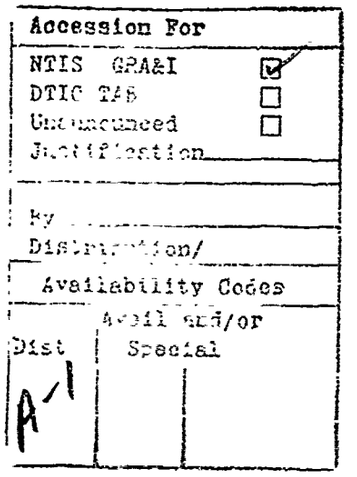
Myra Jean Prelle
Principal Scientist, G110

MITRE Project Approval: Myra Jean Prelle

Myra Jean Prelle
Principal Investigator, 7850

ABSTRACT

We describe a multicomputer run-time executive called the Synchronization Manager (SaM). SaM makes it easier to develop, debug, and enhance multicomputer software because it automatically manages the synchronization required by an object-oriented program to produce the same results as a single-computer execution. We have used SaM to run application programs written in an object-oriented extension of C, called CPM, on a Symult S2010 multicomputer. CPM is described and performance results are presented that suggest the viability of our approach.



ACKNOWLEDGMENTS

The work described in this paper was performed by The MITRE Corporation under a project funded by the Air Force Electronic Systems Division. The project office is at the Rome Laboratory, Griffiss Air Force Base, Rome, New York. The project officer is Lt. C. Blake.

In addition, we thank Thomas J. Brando and Edward H. Bensley for their encouragement and ideas. We thank Lori Phillips for her probing questions that helped us to a successful result that would not have been possible without her concern and dedication. We thank Lt. Cheryl Blake for her careful reading and good suggestions on how to improve this document.

TABLE OF CONTENTS

LIST OF FIGURES

SECTION 1

INTRODUCTION

Performance requirements for modern military application programs dictate the need for more processing-power than is available in conventional single-computer systems. Thus, proposed military systems, such as SDI, JSIP, Joint Stars, and AWACS, plan to use multicomputer systems to meet their needs. The problem with multicomputers is that they are inherently much harder to program than single-computer systems. Programming a multicomputer application may require managing the synchronization among tens, hundreds, or even thousands of independent processes that must be coordinated to solve a single problem. This project seeks to make programming such computers significantly less difficult, thus reducing the cost and risk of using them in military systems.

A major concern is the programmer's ability to manage the synchronization required by a large complex program. Perhaps the interactions among the program elements is very complex, perhaps it makes use of code written by others, or perhaps synchronization at particular points in the program depends on input data. Debugging multicomputer programs is generally more difficult than single-computer programs, because they may exhibit intermittent errors due to slight timing differences when two or more threads of control access the same memory location in an unsynchronized manner. When enhancements are made to a multicomputer program, errors may arise caused by timing differences introduced by the enhancements. In an attempt to address these issues, we are developing a multicomputer run-time executive called the Synchronization Manager (SaM). SaM makes it easier to develop, debug, and enhance multicomputer software, because it automatically manages the synchronization required by an object-oriented program to produce the same results as a single-computer execution. Timing differences have no effect on the results produced by an application program executed on a multicomputer using SaM. In addition, SaM can exploit input data-dependent concurrency that can only be identified at run time.

Object-oriented programming is a good model of computation for distributed-memory, message-passing multicomputers, because it minimizes global information and provides natural communication and synchronization boundaries. When writing programs for a multicomputer, it is a good idea to assign data items and code that will be used together to the same processor. When writing an object-oriented program, the programmer divides the data and the functions to control that data into objects. A side effect of this assignment is that data and functions that will be used together are identified. Thus, object-oriented programming facilitates the automatic mapping of software to hardware performed by SaM. Most importantly, these are a natural part of the object-oriented programming model of computation.

In this paper, we will describe the synchronization manager. The synchronization manager uses future objects and checkpoint and rollback to generate and manage synchronization. Context objects manage method execution, including handling recursive cycles of messages correctly. There are a number of objects which comprise the SaM run-time executive. These objects manage synchronization and communication for the application program. SaM cannot run ordinary C++ application programs; memory management and error handling must be carefully controlled in SaM. However, our application programming language, CPM, is syntactically similar to C++. We have used SaM to run application programs written in CPM, on a Symult S2010 multicomputer. To test the performance of SaM against a truly wide variety of application programs, we developed a synthetic application program. Our performance results suggest the viability of our approach.

SECTION 2

THE BASIC SYNCHRONIZATION MANAGER

The run-time behavior of a program with a given input data set can be represented as a directed graph with cycles (figure 1). The circles in this figure represent objects, and the arrows represent messages. When an object receives a message, the method associated with that kind of message begins executing. The labels on the arrows show the order in which messages are processed when we execute this program on a single computer. Every arrow represents a request for processing. Associated with each request is a reply message that is not shown.

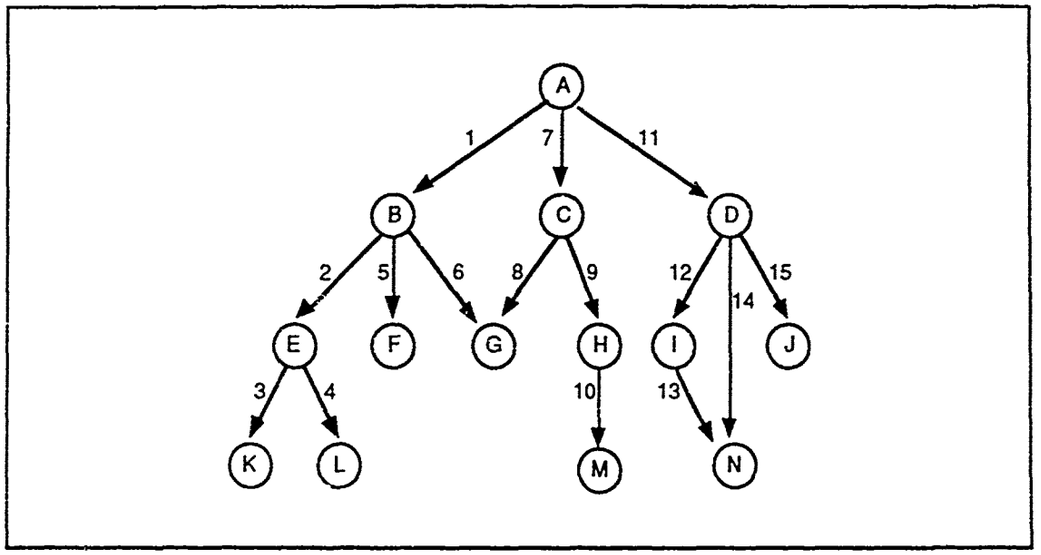


Figure 1. Serial Execution

In SaM, each object in the system that requires synchronization services is associated with an instance of the synchronization manager class. We refer to all instances of this class as *the* synchronization manager.

SaM uses data-driven synchronization and speculative computation to gain concurrency, and checkpoint and rollback to ensure the same results as a single-computer execution. Data-driven synchronization means a process does not block until it needs the result of another process' computation. SaM uses future objects to support data-driven synchronization in basically the same way they are used in the Actor model of concurrent object-oriented programming [Agha:86]. In Actors, ES-Kit [Chatterjee:89], and MultiLisp [Halstead:85], it is the programmer's responsibility to manage futures; in SaM, they are created and managed automatically.

On a single-computer system, every object processes one message at a time in a fixed order (dependent on the input data). On a multicomputer, messages may arrive at an object in a different order, even with identical input data. Speculative computation means that an object processes available messages in the correct order, but without regard for messages not yet delivered, even if those late arriving messages would have been processed earlier in a sequential execution.

To ensure the same results as a single-computer execution, we generalized the Time Warp synchronization mechanism that was developed for distributed object-oriented discrete-event simulation [Jefferson:87]. In both Time Warp and SaM, an object's state is saved, or checkpointed, whenever it processes a message. If an object has processed a message out of order, it is rolled back to the appropriate state, and messages are reprocessed in the correct order. It may be that as a result of processing messages out of sequence, erroneous messages were sent to other objects. During rollback *negative* messages are sent to retract erroneous *positive* messages. When an object receives a negative message, there are three possibilities. If the matching positive message is among the object's unprocessed messages, the two messages cancel each other. If the positive message has not been received yet (this is possible in a network that employs adaptive routing [Chow:87]), the object's synchronization manager saves the negative message so that it can cancel with the positive message when it arrives. If the positive message has already been processed, the object's synchronization manager rolls the object back to the state prior to the one the positive message was processed in. The negative message then cancels with the positive message. A negative message is an *antimessage* for its positive message, and a positive message is an antimessage for its negative message.

In Time Warp, it is the responsibility of the application program to generate timestamps that are used to order messages properly; SaM generates these timestamps automatically. An advantage of both Time Warp and SaM is that deadlocks and races cannot occur. This reduces the difficulty of developing software for multicomputers considerably.

Computation Time

In discrete-event simulation, simulation time is used to determine the order that events (messages) should be processed in, and the application program is responsible for associating a timestamp with each message an object sends. In general-purpose computation there is no sense of time, but there is a sense of order. So we developed a mechanism that works like simulation time to indicate the order that computation events should be processed in. The synchronization manager associates a string of characters with each message an object sends. This character string indicates the order that the message would have been processed in if the computation had been executed

sequentially. Since the role of these character strings is similar to timestamps used in discrete-event simulation, we call them timestamps also.

When an object processes a message with a given timestamp, its synchronization manager appends a character to that timestamp for every message the object sends. In figure 2, for example, the object B receives a message with timestamp "a". The timestamp on the first message that B sends while processing this message is "aa", the timestamp on the second message is "ab", then "ac", and so on. Notice G's synchronization manager can recognize that the message from B should be processed before the message from C because "ac" is less than "ba" lexicographically. However, G speculatively processes whichever message arrives first Before any message is processed, the synchronization manager saves G's current state. Thus, if the "ba" message arrives and is processed before the "ac" message, G can be rolled back to its previous state when the "ac" message arrives so that "ac" and "ba" can be processed in the correct order.

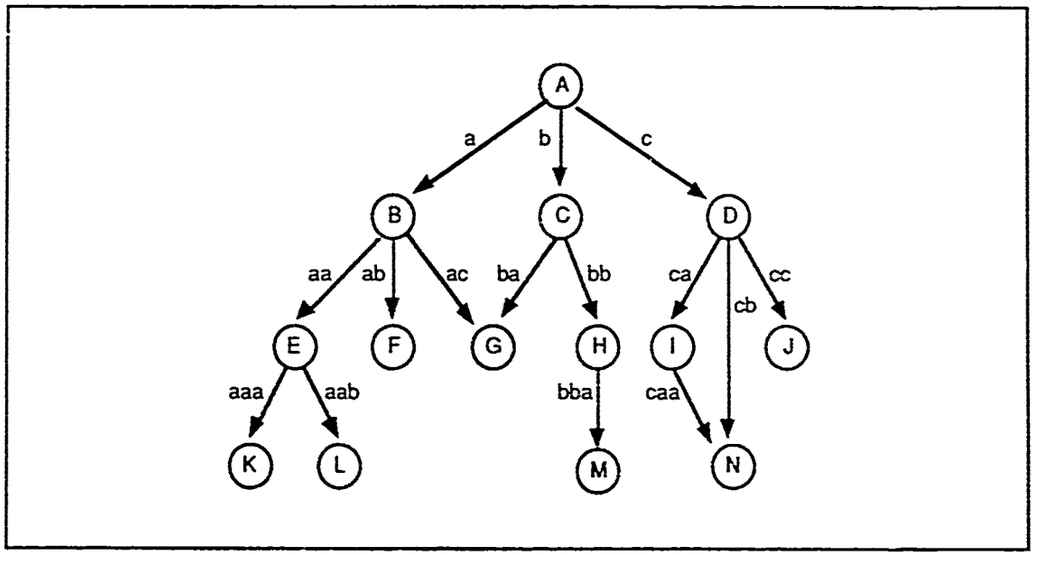


Figure 2. Event Order Synchronization

In general, an object processes one message at a time. Recursive cycles of messages are an exception to this rule. In a recursive cycle, an object must process one message in the midst of processing another message, for example, a recursive cycle can be established directly when an object sends a print message to itself in the middle of executing a method. It is also possible for a recursive cycle to be established indirectly through another object. When a recursive cycle of messages is recognized, an object processes more than one message at a time, but the processing is serialized in the same way as in a conventional sequential execution.

The synchronization manager can recognize when an object has received a message that is part of a recursive cycle of messages, because the timestamp on the object's current state will be a prefix of the timestamp on the recursive message (following any rollback that may be necessary as a result of receiving the recursive message). In figure 3, we see that there are four objects that send messages to M: H, I, J, and N. Suppose that the processing of the message from I to M results in M sending a message to N, and N's processing of that message causes N to send a message to M, that is, a recursive cycle is established. However, let us also assume that none of the other messages to M causes a recursive cycle to occur.

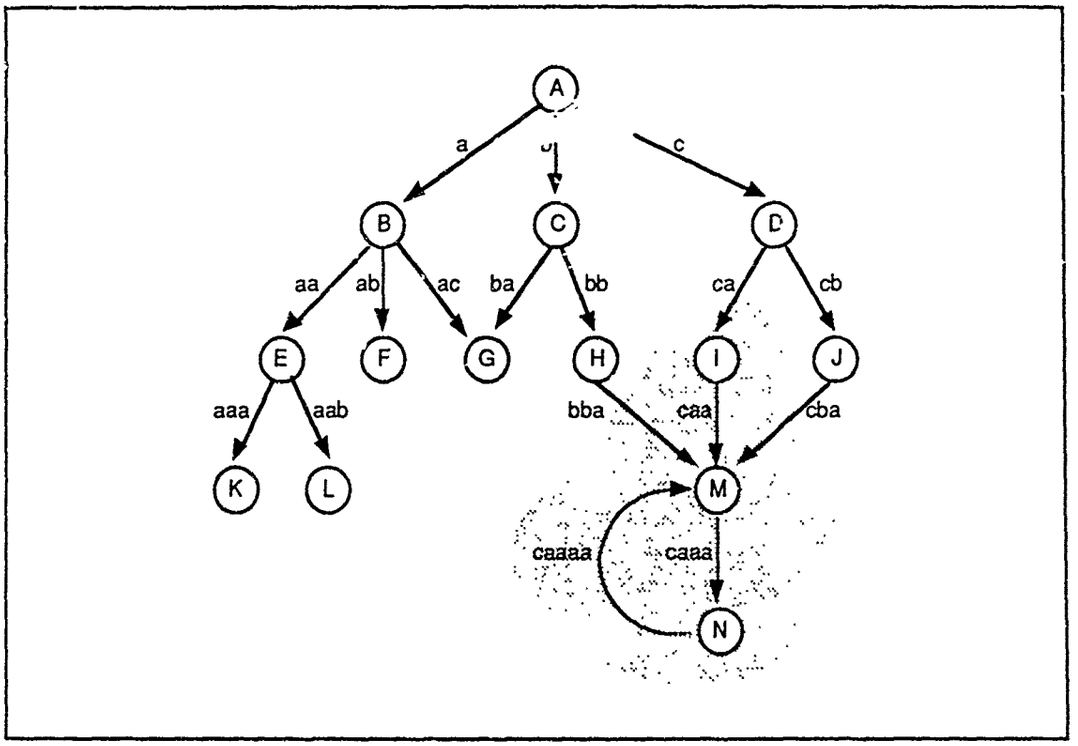


Figure 3. Recursive Message Cycle

M's synchronization manager has to be able to identify three different situations: a message with an earlier timestamp than the one it just processed or is currently processing; a message that indicates a cycle is about to take place; a message with a later timestamp that is not part of a cycle. If the timestamp of the message that M is processing is greater than the timestamp of the incoming message, then M must be rolled back. In the example, if M is processing the "caa" message from I when it receives the "bba" message from H, M must be rolled back.

If the timestamp of the message that M is processing is less than the timestamp of the incoming message, then M's synchronization manager must decide if the incoming message should be processed after the current message has been completely processed or

if a recursive cycle is taking place. In the example, if M is processing the "caa" message from I when it receives the "cba" message from J, M's synchronization manager should keep this message queued until M has completely processed the "caa" message. On the other hand, if M receives the "caaaa" message from N, it must recognize this as a recursive cycle which must be processed before the completion of the "caa" message. M's synchronization manager can recognize that a recursive cycle is occurring because the timestamp of the current message "caa" is a prefix of the timestamp of the incoming message.

The width of a timestamp depends on two factors: the maximum number of messages an object can send in the course of executing a single method, and the maximum depth of a method invocation sequence (for example, A sends a message to B, which in turn sends a message to C. etc.). The latter is equivalent to the maximum number of stack frames on a processor's control stack at any given time in a single-computer execution. The 256 ASCII character set permits an object to send as many as 256 messages during the execution of a single method. For some applications, this may be insufficient. Thus, we have implemented timestamps as an array of short integers (16-bit) also. In this implementation, timestamp comparison is performed by comparing individual elements of the arrays.

Thus, given timestamps A and B ($|A|$ = the length of timestamp A, and $|B|$ = the length of timestamp B), their relationship is defined as follows:

$$A = B \rightarrow |A| = |B| \wedge \forall\, i \in [0, |A|)\ (A_i = B_i)$$
$$A < B \rightarrow |A| < |B| \wedge \forall\, i \in [0, |A|)\ (A_i = B_i)$$
$$\vee\ |A| = 0 \wedge |A| < |B|$$
$$\vee\ \exists\, j \in [0, \mathit{min}\ (|A|, |B|))\ \{A_j < B_j \wedge \forall\, i \in [0, j)\ (A_i = B_i)\}$$

This implementation is similar to the Dewey decimal timestamps employed in ParaTran [Tinker:88].

The short array implementation permits an object to send many more messages in a single method than would really be needed by a reasonable application. Thus, memory to store timestamps is wasted unnecessarily. Furthermore, since most methods only send a few messages, both character string and short array timestamps tend to be used sparsely. A possible way to address this problem is to let each timestamp consist of a bit-string and a length. For each message an object sends it appends two zero bits to the timestamp of the message it is processing and a number of pairs of bits (01, 10, or 11) that indicate the number of messages it sent previously. Suppose A receives a message with timestamp 11, then it sends messages with timestamps:

```
11 00 01                    A's first message sent
11 00 10                    A's second message sent
11 00 11                    A's third message sent
11 00 0101                  A's fourth message sent
11 00 0110                  A's fifth message sent
11 00 0111                  A's sixth message sent
11 00 1001                  A's seventh message sent
11 00 1010                  A's eighth message sent
11 00 1101                  A's ninth message sent
11 00 1110                  A's tenth message sent
11 00 1111                  A's eleventh message sent
11 00 010101                A's twelfth message sent
```

The double zero bits are used to indicate where the breaks between timestamp *segments* occur. For example, in the timestamp 11001101, 11 and 1101 are two segments. Comparison is performed based on comparing segments, just as we compared array elements or characters in strings. We have not implemented this version of timestamps.

Global Virtual Time

A disadvantage of a rollback scheme is that a good deal of memory can be used to save old states and messages. Time Warp uses the concept of *global virtual time* (GVT) to reduce the amount of information that must be kept. The essential idea is that the computation is always moving forward. Thus, there is a simulation time (or, in our case, a computation time) past which the computation can never roll back. This time is the GVT, and ways exist for computing a safe approximation to GVT dynamically while computation is proceeding [Jefferson:87, Samadi:85].

For GVT to be estimated correctly (that is, safely), the timestamps on all the messages in the system must be taken into consideration, even those messages that may be in transit at the time GVT is calculated. To ensure that the timestamp of every message is taken into account, each message is acknowledged. (Shared memory implementations of Time Warp may not require acknowledgements. However, we are not aware of any way they can be avoided in a distributed-memory implementation that supports adaptive routing.)

System-wide GVT is estimated periodically in SaM by requesting application objects to compute their *object virtual time* (OVT). If an application object has no unprocessed messages and no unacknowledged messages, it reports an OVT of infinity. Otherwise, it reports the minimum timestamp on its unprocessed and unacknowledged messages. (Our method for calculating OVT for application objects does not make use of the timestamp on the object's current state. As we will see in the section describing context objects, it is the application object's contexts that keep track of the current time.) The minimum of all the OVT's in the system is selected as the new estimate of GVT, and a message is sent to propagate this value to all objects in the system.

Because GVT is calculated while the computation is proceeding, it is possible that an object on one processor has already computed its OVT before an object on another processor has begun. Suppose A on processor P1 and B on processor P2 are the only application objects in the system. Suppose A is asked to calculate OVT. If A has no unprocessed or unacknowledged messages, it reports infinity for its OVT. Now suppose B sends A a message and receives an acknowledgement before it is asked to calculate its OVT. If B has no unprocessed and no unacknowledged message, it reports infinity for its OVT. Since A and B are the only objects in the system and they both reported an OVT of infinity, GVT is assumed to be infinity, that is, the computation is assumed to be complete. However, A is processing a message. To avoid this difficulty, if an object has already calculated OVT, but a new GVT has not yet been assigned by the system, the object acknowledges all messages it receives with *caution*. If an object has not calculated OVT, since the last time GVT was assigned, it keeps track of the minimum timestamp on any cautionary acknowledgements it receives. When an object is asked to calculate OVT, it considers this value as well as its unprocessed and unacknowledged messages.

We can think of GVT as the commit time of the computation. After a GVT estimate is computed, states and messages with earlier timestamps can be discarded. Speculative computation can cause the application to commit errors (for example, divide by zero) that a sequential execution would not have committed. Eventually, rollback will undo such errors. Thus, application output and errors can only be committed when GVT has passed the point in the computation when they were generated.

SECTION 3

DYNAMICS OF METHOD EXECUTION

Context Objects

In SaM, application messages are packaged inside of *request* messages. When an application object processes a request message, its synchronization manager creates a synchronization managed *context object*. This context object manages the execution of the method associated with the application message that is contained within the request message. A context object's state consists of its application object's instance variables and its method's local variables and arguments. Although we say that during method execution an object sends a message, it is actually the context that sends the message. Similarly, if the value of a future is needed during method execution, it is the context's synchronization manager that sends the message to retrieve the value, and the context's synchronization manager that receives the reply. To support recursion and retraction of replies, a context's state is saved each time it receives or sends a message.

After creating a context, the application object's synchronization manager sends it a *start* message that contains the timestamp and argument values in the request message, and copies of the object's instance variables. When the context processes this start message, it begins executing its method under the control of its own synchronization manager. When the method completes executing, the context's synchronization manager sends its application object a *done* message that contains new values for the object's state variables. Until this done message is received, the synchronization manager does not normally allow an application object to process messages with later timestamps. An exception to this rule is in the case of recursion, as we shall explain later.

Figure 4 illustrates the dynamics of method execution. In this figure, object A receives a request message with timestamp "m" (Q-m). A's synchronization manager creates context A-C0 and sends it a start message with timestamp "m" (S-m). While executing its method, A-C0 sends request messages to objects B (Q-ma) and C (Q-mb). By the way we determine computation time, these messages are sent with timestamps "ma" and "mb." B and C handle their messages similarly to the way A handles its Q-m message. When A-C0 has completed executing its method, its synchronization manager sends a done message (D-mc) to A with new state variable values. When A receives the done message, its synchronization manager updates A's instance variables.

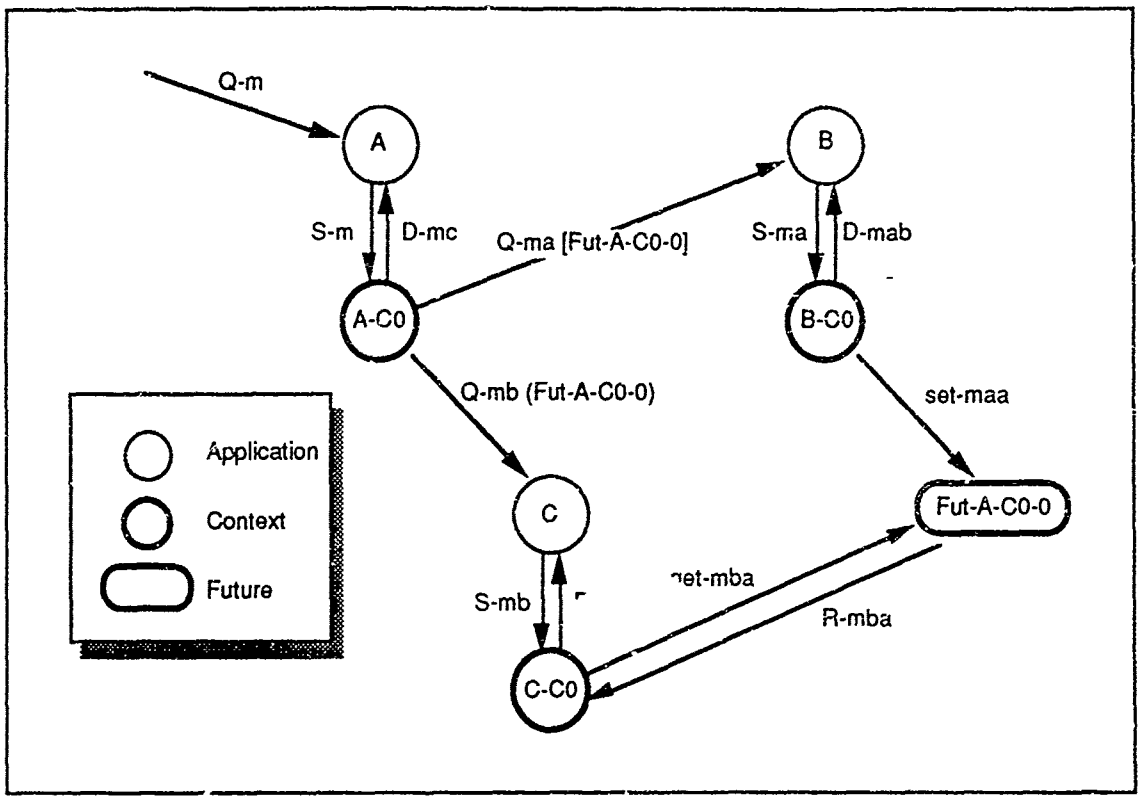


Figure 4. Method Execution

Before a method sends a request message that requires a result to be returned to another object, the synchronization manager creates a synchronization managed future object to hold the result of the computation that will be performed as a result of processing the message. Figure 4 also shows a future object, Fut-A-C0-0, that is used to hold the result of B's computation. Before A-C0 sends the request message (Q-ma) to B, A-C0's synchronization manager creates the future object Fut-A-C0-0. When the message is sent to B, the address of Fut-A-C0-0 is passed in the result field of the message. Let us suppose the value of B's computation must be sent to C as argument. When A-C0 sends the request message (Q-mb) to C, it passes the address of Fut-A-C0-0 as an argument. After processing A-C0's request, B-C0's synchronization manager sends the result of the computation to Fut-A-C0-0 in a *set* message (set-maa). When C-C0 needs the actual value of the future argument, its synchronization manager sends a *get* message (get-mba) to Fut-A-C0-0 and suspends C-C0's method execution until a *reply* message is received. If Fut-A-C0-0 has already received the set message from B-C0 when it receives the get message from C-C0, its value is returned immediately to C-C0 in a reply message (R-mba). If Fut-A-C0-0 has not received the set message from B-C0, Fut-A-C0-0's synchronization manager postpones sending a reply to C-C0 until Fut-A-C0-0's value has been set. When C-C0 receives the reply, its synchronization manager sets the variable that holds the pointer to the future to the value contained in the reply.

The mechanism described thus far is not capable of handling recursive cycles, which require that an object suspend the execution of one method, receive another message, and execute to completion the method that correspo..ds to the new message, before it continues the first method. With recursion, an object has more than one context active at the same time. An application object's synchronization manager maintains a stack of *active contexts* as part of the object's state that is used to serialize the computation correctly when a recursive cycle occurs. The stack has a context for each level of recursion, as well as one for the original request message.

A recursive cycle occurs if A is sent a request message by one of its own contexts or by some other object's context as a result, direct or indirect, of a request message sent by a context of A. Figure 5 illustrates how recursive method execution is performed. Suppose A's context A-C0 sends a request message with timestamp "ma" to A. Ordinarily, A's synchronization manager would not permit it to process a request message with a timestamp greater than the current request message until an appropriate done message had been processed. But in this case, A's synchronization manager can tell a recursion has occurred, because the current message with timestamp "m" is a prefix of the new message's timestamp, therefore, the Q-ma message must be processed before the Q-m message can complete.

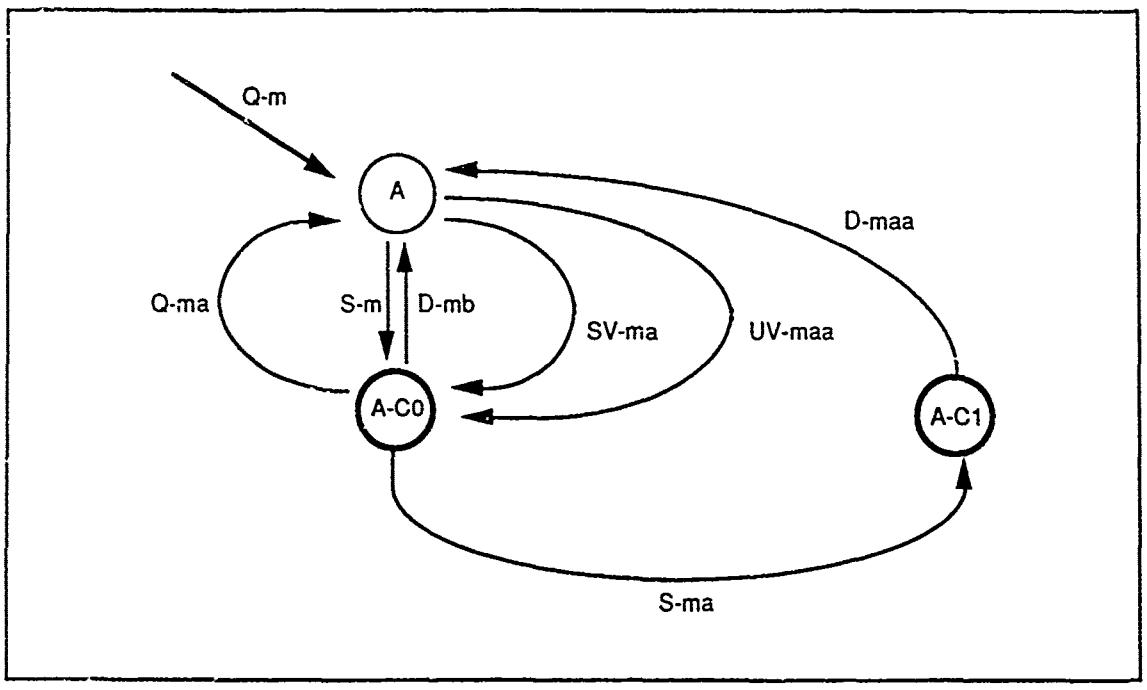


Figure 5. Method Execution With Recursion

In the figure, when A's synchronization manager receives the recursive Q-ma message, it creates context A-C1 and pushes the new synchronization managed context onto A's current state's active context stack. Then it sends the context that was previously on the top of the stack, A-C0, a *send_values* message with the timestamp "ma" (SV-ma) and a

pointer to A-C1. When A-C0's synchronization manager receives the SV-ma message, it rolls back (if necessary) to A-C0's state immediately after sending the Q-ma message. It then sends A-C1 a start message. This message contains copies of A's state variables with the values they had at the time that A-C1 sent the request message that caused the recursion. This ensures that the method associated with A-C1 has the correct initial values for A's state variables. When A-C1 finishes executing its method, its synchronization manager sends A a done message with new values for the state variables. A's synchronization manager pops A-C1 off A's active context stack and forwards the new values to A-C0 in an *update_values* message (UV-maa). A-C0's synchronization manager blocks A-C0's method execution in its "ma" state until it receives the UV-maa message from A. Then it resumes A-C0's method with its copies of A's state variables correctly reflecting the complete processing of the recursive message.

In SaM, we implemented a test that checks whether a method is sending a request message to its own application object. When such a message is sent, the context's synchronization manager puts it in a recursion wait state. Since it is fairly common for a method to initiate the execution of another of its object's methods, this test reduces the frequency that rollback occurs. However, rollback due to recursive cycles may still be necessary if the recursion is initiated indirectly.

Context objects calculate OVT differently than application objects. If a context object has finished executing its method, has no unprocessed messages, has no unacknowledged messages, and has received no cautionary acknowledgements, it reports an OVT of infinity. Otherwise, it reports the minimum timestamp on its unprocessed messages, unacknowledged messages, cautionary acknowledgements, or its current state (usually the timestamp on the next message it is going to send).

Context Synchronization Manager Services

In addition to managing a context object's synchronization (state saving, rollback, unprocessed and unacknowledged messages) a context object's synchronization manager provides services to a context as it is executing. The synchronization manager application language translator inserts invocations for synchronization manager services into the application method code during translation. When a context object needs to send an application message to another object, it invokes its synchronization manager passing the address of the object to which the message is to be sent and the information necessary to invoke the correct method of the receiving object. Its synchronization manager creates a synchronization managed future object (if necessary) and packages the application message in a request message to the other object.

When a context object needs to create another application object, the context object invokes its synchronization manager passing the information necessary to create the new

object. Its synchronization manager creates a synchronization managed future object and packages the information in a *make_instance* message to a creator object.

When a context object needs to output information, the context object invokes its synchronization manager passing the information to be printed. Its synchronization manager packages the information in a *print* message to an output object.

When a context object needs a future resolved to a non-future value, it invokes its synchronization manager passing the address of the future object that contains or will eventually contain that value. Its synchronization manager sends a get message to the appropriate future object.

When a context object completes execution, it invokes its synchronization manager passing the result of its computation (if appropriate). Its synchronization manager sends the result to the appropriate future (if necessary) and sends a done message to the context's application object.

SECTION 4

SAM RUNTIME EXECUTIVE

Any number of application, context, and future objects (embedded in their associated synchronization managers) may be assigned to any processor in the system. In addition, each processor has a harness object, a creator object (embedded in its synchronization manager), and a gvt_controller (figure 6). Processor zero has some special objects assigned to it including a gvt_master and a start object and context (embedded in their synchronization managers), as shown in figure 7. In addition, there are message objects which are not shown in either figure. The role these objects play in the SaM runtime executive will be explained in this section.

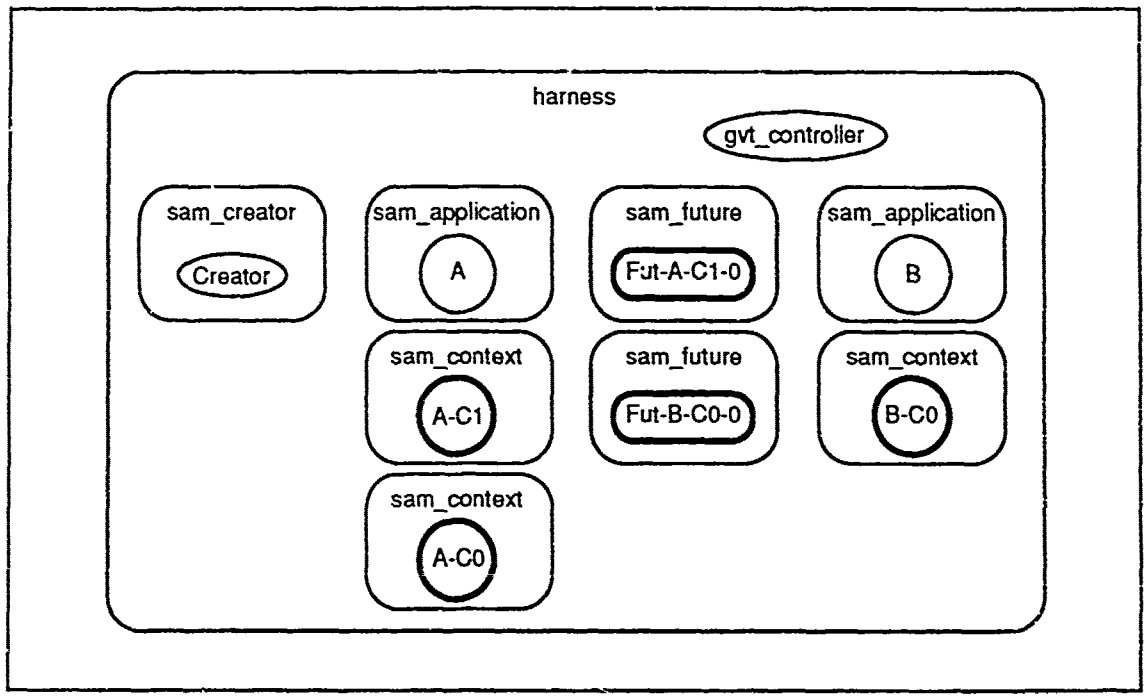


Figure 6. A Processor Other Than Processor Zero

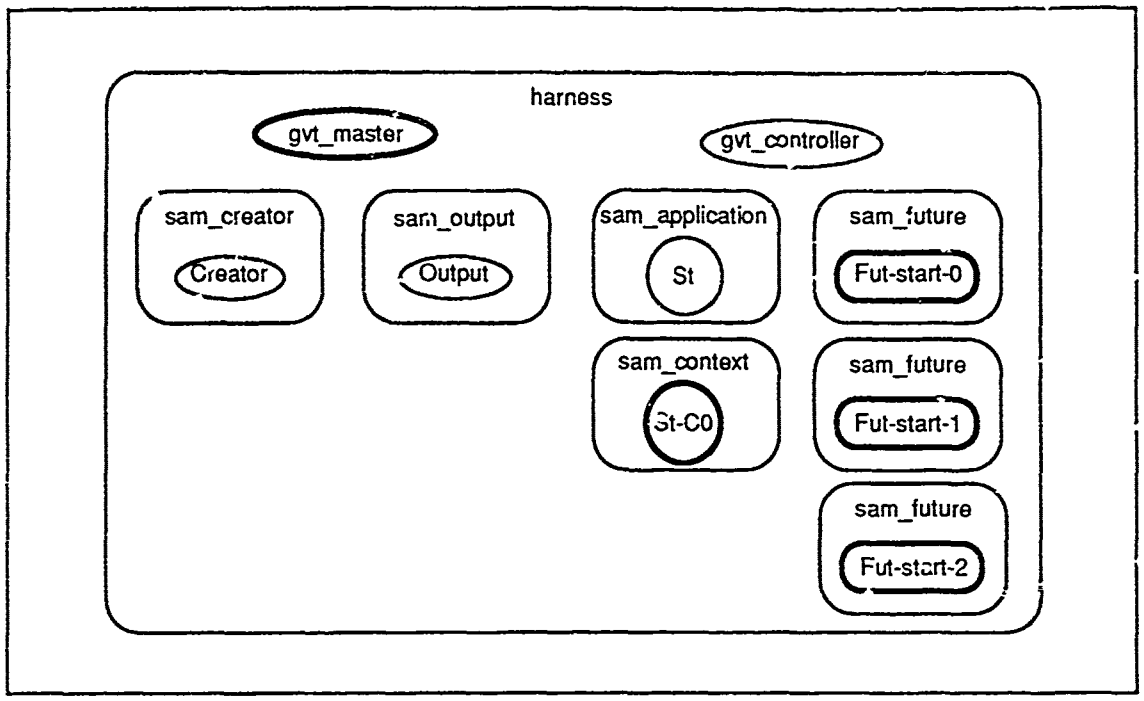


Figure 7. Processor Zero

Sam Objects

Synchronization manager objects, *sam objects*, have instance variables that include: input_queue, current_state (which includes the state variables, as well as the message the object processed while in the state and the messages the object sent while in the state), old_states, and unacknowledged message list. Sam objects manage the synchronization for their respective application, context, future, creator, and output objects. The functions of the synchronization managers for application and context objects have been explained above. Creator, future, and output objects process messages differently from application and context objects. Associated with each of these are specializations of the basic synchronization manager class for application objects. Every synchronization managed object (application, context, future, creator, and output) has a unique *global object address* associated with it.

Message Objects

A message is an object that contains fields that include: the global object address of the object to which the message is sent, the global object address of the object that sent the message, the receive timestamp (indicating the order that the message is to be processed in by the receiver of the message), the message type (request, done, start, send_values, update_values, reply, get_value, set_value, and make_instance), the information field, and the result field. The receiving object's synchronization manager interprets the information and result fields differently depending on the message type. For example, in

a request message, the information field contains the name of the application message and the values of the arguments that should be associated with that message. If the result of a method invocation is required by the requesting object, the result field contains the address of the future object that is to be sent the value when method execution is complete, otherwise it is NULL.

In addition to these fields, each message also contains a sign field. The sign field is used to indicate whether the message is a positive message or a negative message. Each message also contains an acknowledgement field that indicates whether the message is an acknowledgement or not.

One message may be considered an *antimessage* for another message, if the two messages contain the same information in the corresponding fields except for the sign field. We could check that every field (except the sign field) of two messages are identical before deciding whether they were antimessages. However, this is not necessary if we add another field to the message called the rollback count. Every object's synchronization manager has a rollback count that it maintains. Every time an object is rolled back its synchronization manager increments the rollback count. The object's synchronization manager includes the value of the rollback count in each positive message it sends on behalf of the object. Each state of an object has a copy of the messages the object sent while in that state, so that negative messages can be sent if the state is rolled back. However, a negative message need not be a complete copy of the message; both the information and result fields can empty. This is possible because the receiver, sender, timestamp, and rollback count are sufficient to associate a negative message with its matching positive message. Using similar reasoning acknowledgements and messages in the unacknowledged message list can have empty information and result fields.

Harness Objects

The multicomputer implementation of the synchronization manager has a harness object on each processor. The harness objects manage the other objects in the system. They manage communication among objects, placement of new objects, routing of messages between processors and among application objects, and scheduling of activities in the system.

Communication among synchronization managed objects is mediated by the harness objects. When an object is to be created, the harness object on the requestor's processor decides on which processor the object will be created. Currently, each harness object directs creation requests to all processors (except processor zero) in round robin fashion, independently from other harness objects. A global object address includes the identity of the processor on which the object was created. Since objects do not move from the processors on which they were created, this information is sufficient to allow harness

objects to route messages to the processors that have the objects to which they are directed.

The harness object on each processor executes a loop. In the first step of the harness loop, messages are taken from the processor's input queue and delivered one by one to the appropriate object's synchronization manager's input queue. If the message is positive and has a receive timestamp less than or equal to the object's current state's timestamp or the message is negative and matches a message that was previously processed by the object, then the object's synchronization manager rolls the object back to a state with a timestamp less than the message's timestamp. If necessary, the object's synchronization manager sends negative messages and/or cancels the matching positive and negative message pair. Otherwise the message is inserted into the synchronization manager's input queue.

In the second step of the harness loop, each object's synchronization manager is given an opportunity to handle a message from its input queue. Before handling a message, an object's synchronization manager saves the object's current state on its old state stack. Different kinds of messages are handled differently by the object's synchronization manager. In the case of application request messages that cause methods to be executed, the application object's synchronization manager creates a context object whose responsibility it is to actually execute the method associated with the request. In the case of replies from future objects, the context object's synchronization manager changes the value of the variable that points the future to the value of the computation that the future represents.

In the final step of the harness loop, the harness selects one of its contexts and allows it to execute part of its method. The context selected is the one with the lowest timestamp that is not blocked. This context executes until it completes executing its method, blocks, or has exceeded its allotted number of *execution steps*.

A context will block if it needs the value of a future, sends a message to its application object (by executing a send self message), or enters an error state. Because messages are processed speculatively, values of variables may be the wrong type or have inappropriate values when a statement in a method is executed. This causes the context to enter an error state. Execution is blocked until the object rolls back or GVT passes the timestamp on the error state. In the latter case, a real application program error has occurred (just as it would have in a sequential execution), and the program must be aborted.

A context is allowed to execute its method until it requires service (sending an application message, resolving a future, etc.) from its synchronization manager. This is called an execution step. The harness decides how many execution steps a context may take before control returns to the harness. In the current implementation the number of execution steps is arbitrarily set to ten. A very high setting would allow contexts to take

as many steps as they are able to perform before checking for newly received messages. Such a policy could result in larger numbers of unnecessary rollbacks. A very low setting forces the context state to be saved more frequently. We have not performed any studies to determine what the preferred setting for the number of execution steps should be.

At this point the harness may choose to go to the first step of its loop or, if the context with the lowest timestamp is blocked, allow the context with the next lowest timestamp to execute. In this way messages with earlier timestamps are processed before messages with later timestamps, at least locally.

The harness on processor zero has an additional step in its loop. In this step, if a sufficient amount of time has passed (one second in our current implementation) since the last time GVT was calculated, the harness activates the gvt_master.

Creator Objects

Each processor has a creator object associated with it. The purpose of a creator object is to free context objects from the task of managing the creation of application objects required by the computation. The application language translator automatically translates a user invocation of the function "make_instance" to a make_instance message to a creator object. In this way, a context object may continue its computation until it needs to send a message to the newly created object.

When an application object is to be created, the synchronization manager of the context requesting the creation creates a future object on its local processor. This future object will ultimately hold the global object address of the new application object. The context's synchronization manager gets a global object address from its local harness to associate with the new future object. Then it asks its harness for the global address of an appropriate creator object (as explained above in the section describing harness objects). Next it sends a make_instance message containing the future object's global object address and the information necessary to perform the application object creation to this creator. The context may then continue processing using the future.

When a creator receives a make_instance message, it gets a new unique global object address from the harness on its processor, creates the new application object, and sends a set-value message containing the global object address of the new application object to the future object specified in the message.

Creator class objects have associated synchronization manager objects that are refinements of the general synchronization manager class. It is not necessary for creators to be rolled back when they receive make_instance messages out of sequence or negative make_instance messages. When a creator receives a negative message for a message it has processed, its synchronization manager removes the associated state from the old

state stack and cancels the positive and negative message pair—the creator is not rolled back and does not send antimessages or reprocess input messages, except for retracting those messages sent when the cancelled make_instance message was originally processed.

Creator objects do not use contexts to process make_instance messages, as application objects do, but rather process make_instance messages directly. When an object is created, and the make_instance is later retracted (via a negative message to the creator object), the superfluous object continues to exist, in case other objects have sent messages to it. Eventually, any messages sent to the superfluous object will be retracted, and the object will be in its initial state with no messages in its input message queue. When the make_instance is retracted, the creator could send the object a *free* message with the same timestamp as the erroneous make_instance. When GVT passes the time on the free message, the space occupied by the object could be freed. (Our current implementation does not handle freeing objects.)

Future Objects

Future objects are similar to ordinary application objects. However, rollback is handled differently and they do not use contexts to process messages, but process set-value and get-value messages directly. Thus, they have special synchronization manager objects associated with them. A future receives at most one unretracted set_value message, but may receive many legitimate get_value messages. The get_value messages always have timestamps that are greater than the set_value message. A future's synchronization manager holds any get_value messages it may have received in its input queue until the future has received and processed a set_value message.

A future is rolled back when it receives a negative message for a previously processed set_value message. However, it is not necessary for a future to be rolled back when it receives a get_value message out of sequence or a negative message for a get_value message. When a future receives a negative message for a get_value message it has processed, its synchronization manager removes the associated state from the old state stack and cancels the positive and negative message pair—the future is not rolled back and does not send negative messages or reprocess input messages, except for retracting those messages sent when the cancelled get_value message was originally processed.

Output Object

The output object receives messages containing text to be printed to standard output and redirects it to a file. Any other kind of output that a program might produce (for example, file output to a user specified file, instructions to a graphical display, database, or an external device) could be handled in a similar way, by using another type of output object. The output object's synchronization manager does not allow it to process a

message until GVT has passed the time given in the timestamp of the message. Until the system has committed to a GVT, the application program output may be retracted. Since the output object does not process messages until GVT has passed their timestamps, there is no chance that it will be rolled back. Therefore, it is not necessary for the output object's synchronization manager to save states. The output object does not use contexts to process its messages, but processes its messages directly. The computation is considered complete when all processors have returned infinity for their processor virtual time and the output object has no messages to process.

Start Object

Every application program has a special start_class. The main routine of the program is a method of this class. In the multicomputer implementation, code is loaded on each processor. A main routine begins executing on each processor which initializes the harness object on that processor. As part of harness initialization, each harness creates a creator object and a gvt_controller object. The harness on processor zero also creates an output object, a gvt_master, and a start object based on the start class. Harness zero inserts a run message in the start object's synchronization manager's input queue. Then the harnesses begin executing their processing loop. When this message is processed by the start object, a context is created to manage the execution of the main routine of the application program.

GVT_Master and GVT_Controller Objects

The harness invokes the gvt_master based on the last time GVT was calculated. If the last GVT computation is complete, a new GVT calculation begins when the gvt_master sends *calculate_gvt* messages to each gvt_controller. (The harnesses handle delivery of these messages.) The gvt_controllers asynchronously poll the objects on their respective processors and send the minimum timestamp calculated to the gvt_master. When all controllers have reported, the master sends the minimum timestamp received in an assign_gvt message to each controller. The controllers then assign the new GVT to each of the synchronization managed objects on their processors. Each object drops states with timestamps less than the newly assigned GVT.

SECTION 5

APPLICATION LANGUAGE

SaM cannot run ordinary C++ application programs, because memory management and error handling must be carefully controlled in SaM. Our language, CPM (C Plus Minus—C plus objects, minus pointers), is syntactically similar to C++. Programs written in CPM can be executed either sequentially without the synchronization manager or in parallel using SaM. If the program is to be executed sequentially, a translator is used to translate object-oriented CPM code to ordinary C code. If the program is to be executed using SaM, a different translator is used to translate the user's code into a form that will allow SaM to manage its execution.

CPM supports two different types of classes: concurrent classes and local classes. Concurrent objects may be distributed physically in the system. Their synchronization is managed by SaM, that is, each of them has a synchronization manager object associated with it. Concurrent objects have object addresses associated with them. When a concurrent object is passed as an argument, its object address is passed. Circular referencing among concurrent objects is supported. When we spoke of application objects previously, we were describing concurrent application objects.

A local object may be the value of an instance variable of a concurrent object or another local object. A local object is always part of the state of one and only one concurrent object. Thus, their synchronization is managed by their concurrent object's synchronization manager. When a local object is passed as an argument, it is passed by value. The translator automatically generates copy functions to take care of packaging up local object arguments. Pointers to local objects and circular referencing among local objects is not supported (if circular referencing is required, concurrent objects must be used instead).

All basic C data types are supported such as int, char, long, float, double, and statically allocated arrays. In order to make building application programs easier, we are implementing a library of commonly used local classes. Another reason for building these local classes is that part of their implementation is outside the scope of the application language as defined and needs special hand coding. Among the local classes to be implemented are strings, linked lists (parameterized by type), and doubly linked lists (also parameterized by type). These classes will also be robust, checking for errors and boundary conditions. Global constants are allowed in the system, but not global variables. Global variables are not safe since their synchronization is not managed by SaM.

Because SaM allows objects to process messages out-of-order with respect to a sequential execution, errors may occur in a SaM execution which would not have occurred in a sequential execution. If the error is caused by processing messages out-of-order, it must be possible to recover from the error by simply rolling back when the appropriate message or messages finally arrive and are processed in the correct order. On the other had, if the error is a real application error (that is, it would have occurred in a sequential execution), then it can be committed to when GVT reaches the timestamp on the state when the error occurred.

For example, errors that can be trapped by the system (divide-by-zero) send a signal to the executing process. When an error signal is caught, the context is put in an error state. When a context is in an error state, its synchronization manager will accept messages but will not allow the context to execute. Eventually either a message will be received that causes the context to be rolled back to a non-error state, or GVT will reach the timestamp of the error state and the computation will be aborted.

Because an error may be an artifact of speculative computation and not the application program, we cannot allow application behavior to endanger the run-time system or other parts of the application program. Since the application and SaM share the same memory, if we allowed pointers to local memory in our language as C and C++ do, it would be possible to write into memory that is not part of the object's currently executing environment. If this type of error occurred, it would not be corrected by rollback. Thus, each object's state must be managed separately. For arrays, index out-of-bounds errors are captured before they occur.

Interactions among concurrent objects are controlled by the SaM runtime system. The interface points between the application method execution and the run-time system include: creating an instance of a new concurrent object, sending an application message to a concurrent object, printing a result, resolving a future, and completing execution which may include sending a return value to a future.

To support recursive cycles of messages, error handling, and future processing, it must be possible to suspend and resume method execution (possibly to a previous state) when appropriate. We would like to treat method execution as a light weight process. Since LWP libraries are not available on the iPSC/2 or the S2010 (and there are bugs in the Sun LWP code), we implemented a mechanism that is similar to light-weight processes using *setjmp* and *longjmp* system calls. To save the state of an executing method, we must save all registers, local variables (the stack), instance variables of the object, and method arguments. *setjmp* suffices to save all registers, but does not take care of saving anything on the stack. We added code to save the appropriate amount of stack as well as instance variables and method arguments to resume execution. The saved state of a method execution is called an environment.

The interface between the application language and the runtime system is controlled by code inserted by the preprocessor into the application code. This interface code calls the appropriate synchronization manager services. The context's synchronization manager performs the appropriate service and then determines whether or not to resume method execution immediately. If method execution cannot be resumed immediately, a copy of the environment is saved. Since an application message sent by a context can lead to a recursive cycle of messages being established, a copy of the environment is saved when a message is sent to another concurrent object, even if method execution is resumed immediately.

SECTION 6

SYNCHRONIZATION MANAGER PERFORMANCE

In general, the *granularity* of an application is the average amount of computation performed per communication event. Speedup is obtained in parallel computation by having more than one processor perform useful work at the same time. To obtain this speedup, control information needed to activate the necessary functions as well as data (possibly even the functions themselves) may have to be communicated to the appropriate processors. Every multicomputer system requires a certain amount of overhead for each interprocessor communication. A parallel machine execution outperforms a sequential execution when the granularity is sufficiently large to overcome the overhead. In parallel computation, granularity defines the opportunity for parallelism. More parallelism may be exploited if the granularity of the application is small. However, if an application program's granularity is too small compared to the communication overhead of the multicomputer system on which it is executed, speedup may be minimal because application processing may be overwhelmed by communication overhead.

To test the performance of the synchronization manager against a truly wide variety of application programs, we developed a synthetic application program. We can use this program to generate applications that vary in their object referencing and creation characteristics, and their granularity, which we define as the amount of computation they perform per message processed. Essentially, a synthetic application is simply code to control the activities of a program, a pseudo-random number generating function, plus a granularity. The *application control code* uses a pseudo-random number generating function to decide the relational dependencies among the objects that are created initially, what activity (sending messages, creating objects, or doing neither) an object performs in response to receiving a message, how many times it performs the activity, and how the relational dependencies among the objects change as the computation proceeds. In addition to these activities, each time the application control code receives a message, it calls a function. The purpose of this function is to allow us to control the amount of computation the synthetic application performs per method, that is, the granularity.

When the granularity is zero, only the application control code is executed. When the granularity is set to g the function performs g seconds of pseudo-computation. The application set consisted of 54 basic programs with different application control codes. Figure 8 shows the number of objects dynamically created during the execution of each of the basic synthetic applications when they are executed on a single-computer. Figure 9 shows the number of methods invoked during the execution of each of the basic synthetic applications when they are executed on a single-computer. Each of these basic synthetic applications was executed in our tests at seven different granularities: 0, 0.1, 0.2, 0.4,

0.6, 0.8, and 1 second. Thus, there were a total of 378 synthetic application programs used in our study. The application output of each of these 378 programs produced the same values as the equivalent sequential executions.

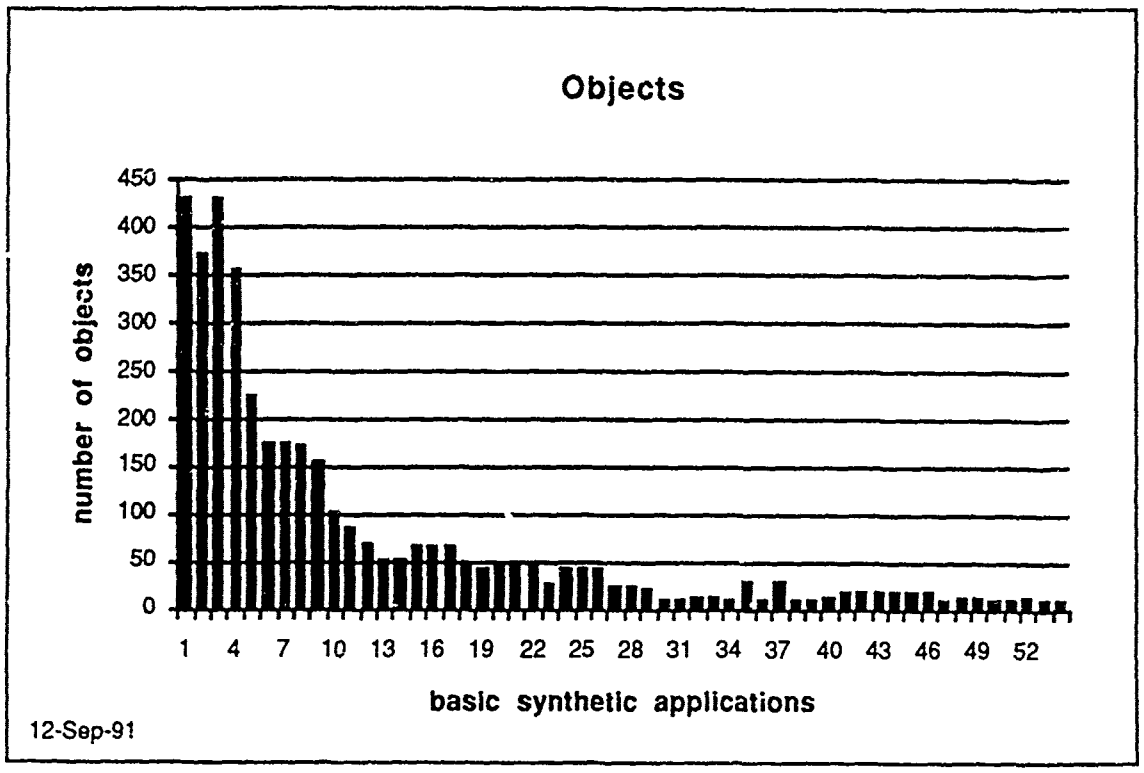


Figure 8. Number of Objects

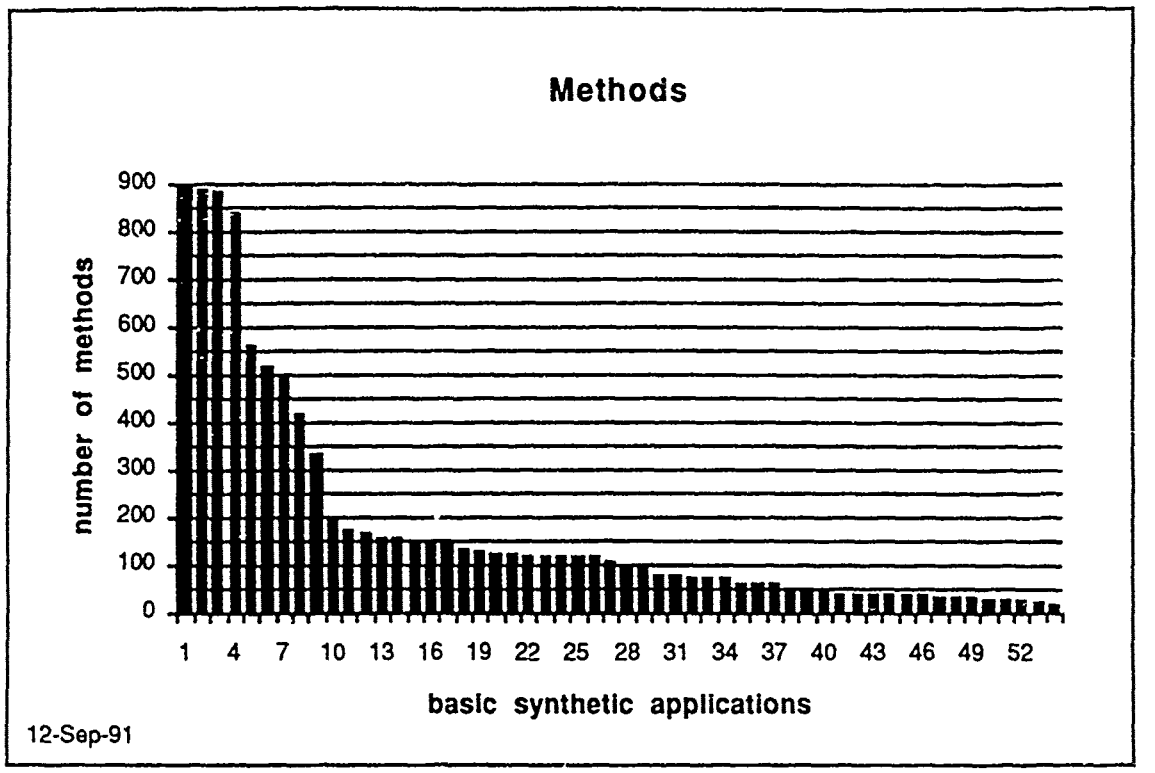


Figure 9. Number of Methods

It is important to understand that these sequential applications were not ideal applications for parallelization. An ideal application for parallelization would have many computations that could be performed independently. Synchronization among the objects would be relatively rare. An indication of how interdependent the computations of the objects in the system are can be inferred from the number of rollbacks that occurred and the number of states rolled back as the computation is executed using the synchronization manager. Figure 10 shows the number of states rolled back when the basic synthetic applications were run on eight nodes using a granularity of 0.1 seconds (0.1 seconds of pseudo computation per method).

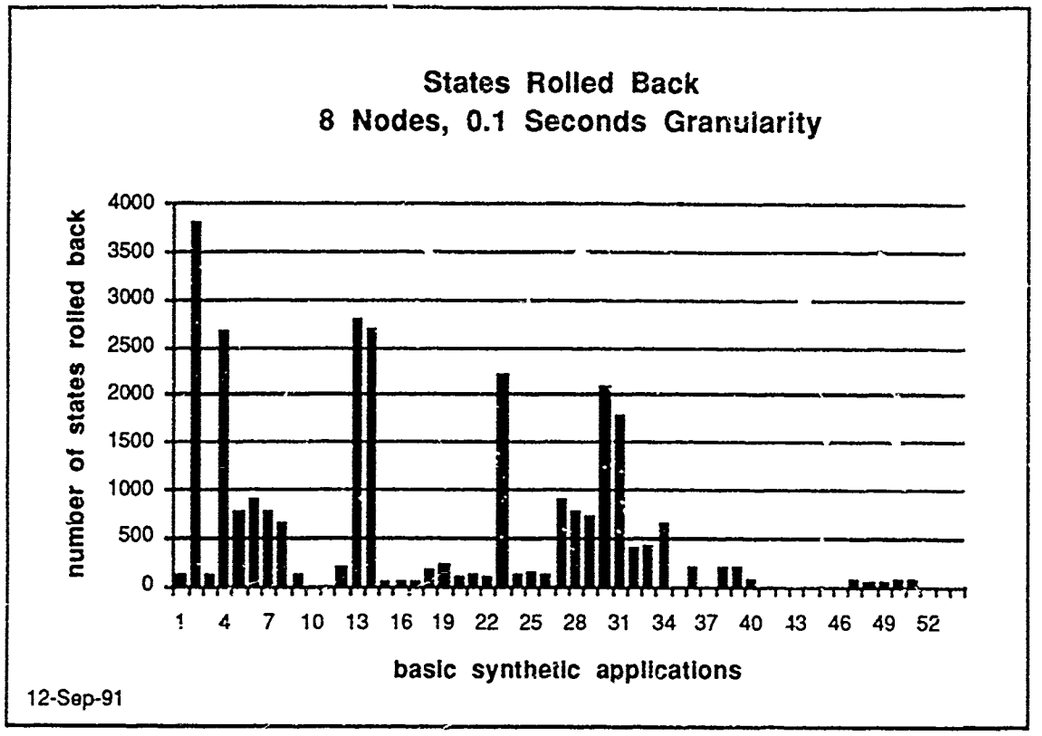


Figure 10. States Rolled Back—8 Nodes, 0.1 Seconds Granularity

Rollbacks may be caused by positive messages received out of order, *positive rollbacks*, or by negative messages that have been sent to retract erroneously sent positive messages, *negative rollbacks*. Figure 11 shows the number of positive and negative rollbacks. Since one rollback may cause many states to be rolled back, the number of rollbacks is smaller than the number of states rolled back. Further, one positive message processed out of order can cause more than one negative message to be sent to retract the messages that were sent erroneously. Thus, it is not surprising that the number of rollbacks that were caused by negative messages is greater than those caused by positive messages.

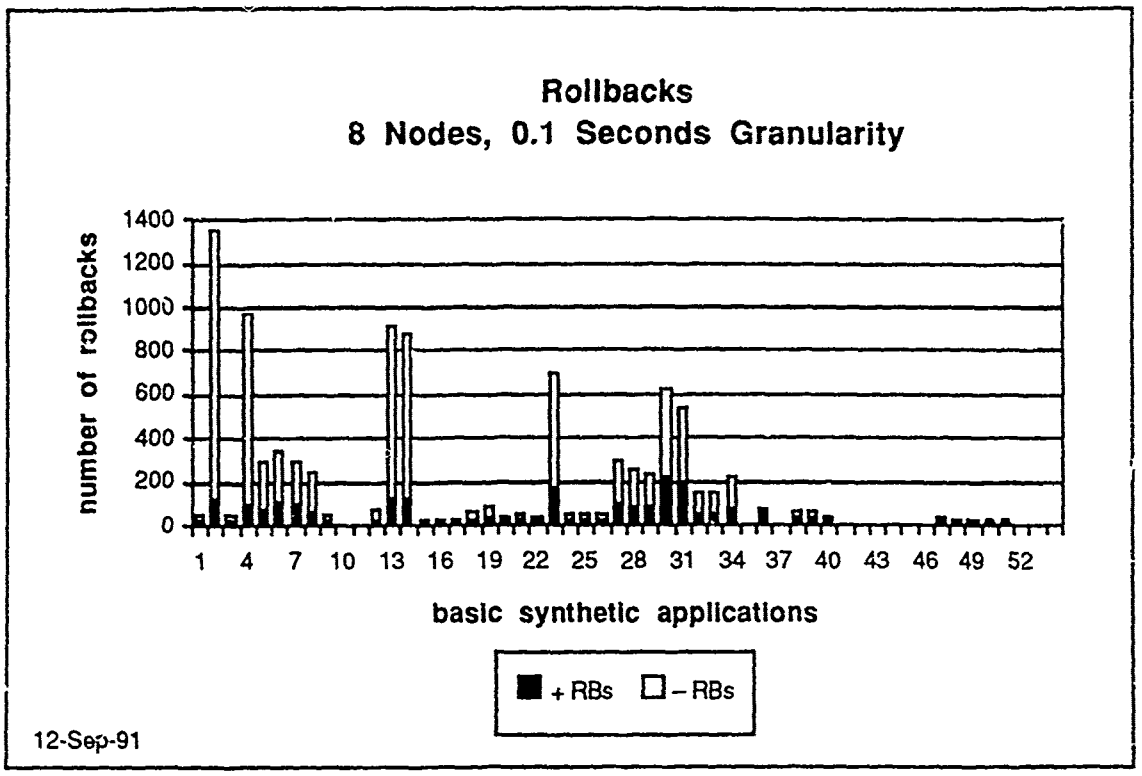


Figure 11. Rollbacks—8 Nodes, 0.1 Seconds Granularity

We ran each of our synthetic applications sequentially on a single Symult S2010 node without the synchronization manager, and in parallel on four, eight, twelve, and sixteen S2010 nodes using the synchronization manager. Runtime was measured using the clock() function in the C library. In the sequential execution, clock() is invoked at the start of main() and again at the end of main(). The runtime of the program is the difference between these two invocations of the function. In the SaM execution, clock() is invoked initially when the harness is initialized on processor zero. It is invoked subsequently when harness zero receives a stop message from the gvt_master indicating the computation, including all application output, is finished. We averaged the runtime values across the seven different granularities and obtained the results shown in figure 12. In the graph in figure 12, the x-axis represents the granularity in seconds of computation per message processed. The y-axis represents the sequential execution runtime divided by the parallel execution runtime, that is, the *speedup*. The place where the curves cross the line y = 1 indicates the granularity where the four, eight, twelve, and sixteen processor executions begin to outperform the sequential computation.

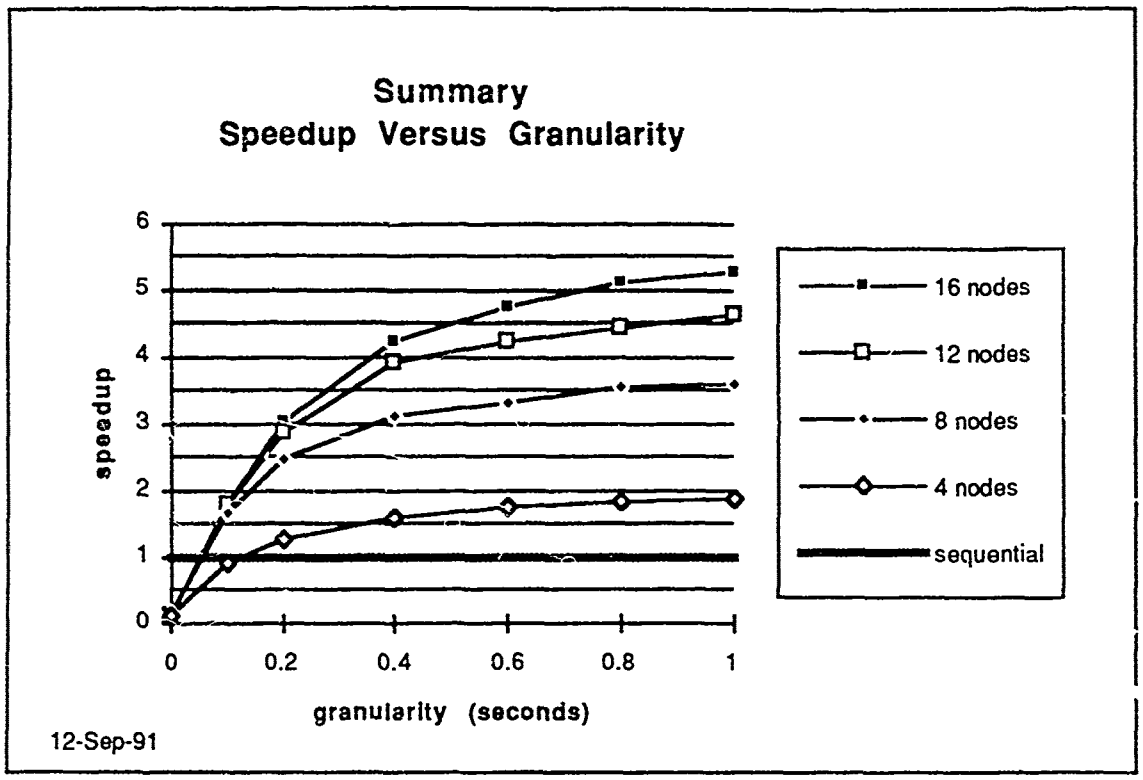


Figure 12. Summary Speedup versus Granularity

Another way to look at this data is to consider the amount of speedup that was obtained for applications with different granularities. Figure 13 shows speedup of applications with different granularities versus the number nodes used.

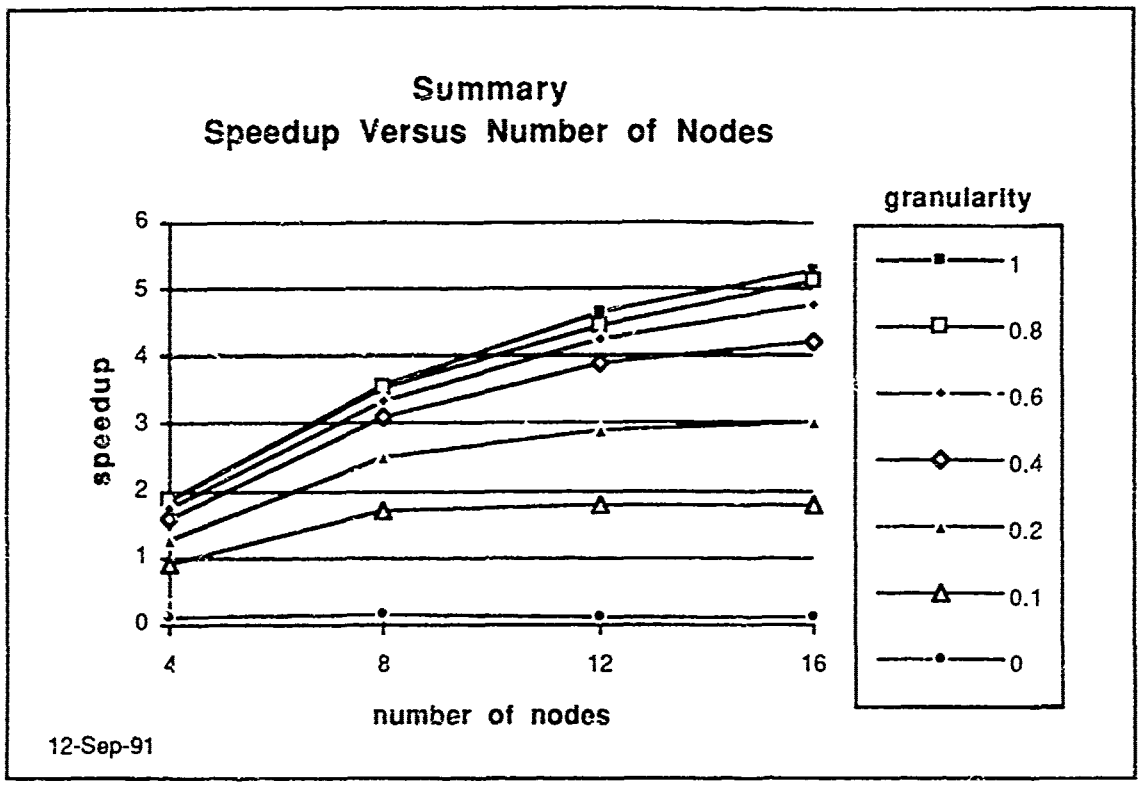


Figure 13. Summary Speedup versus Number of Nodes

It is important to realize that the limit of the speedup function does not approximate the amount of inherent parallelism available in an application. All of the applications have some start-up and finish-up code that must be performed sequentially. In addition, some of the applications in the set involve recursive cycles, which can only be executed sequentially. Thus, it is not surprising that the amount of inherent parallelism available on the average is not higher, that is, closer to the number of available processors.

As we expected, the runtimes of the sequential executions can be approximated by a linear function of granularity. From figure 14, it appears that the runtimes of each of the parallel executions can be approximated fairly well by a linear function of granularity also. This linear approximation makes sense, because the synchronization manager only affects an application when an object sends or receives a message. In general, increasing the amount of computation done per method, that is, the granularity, increases the amount of synchronization manager overhead only slightly. This slight increase in overhead can be attributed to the fact that GVT calculations are done approximately every second and the harness management functions (for example, to check the processor input queue for newly received messages) are invoked more frequently. Thus, the longer a program runs, the more GVT calculations are performed. However, as can be seen from the graph, the GVT calculation and the harness management functions are relatively inexpensive operations. The increase in synchronization manager overhead for longer running computations might be more significant if rollbacks were deep enough to cause lengthy

computations to be redone. On the average, this appears not to be the case for this set of applications.

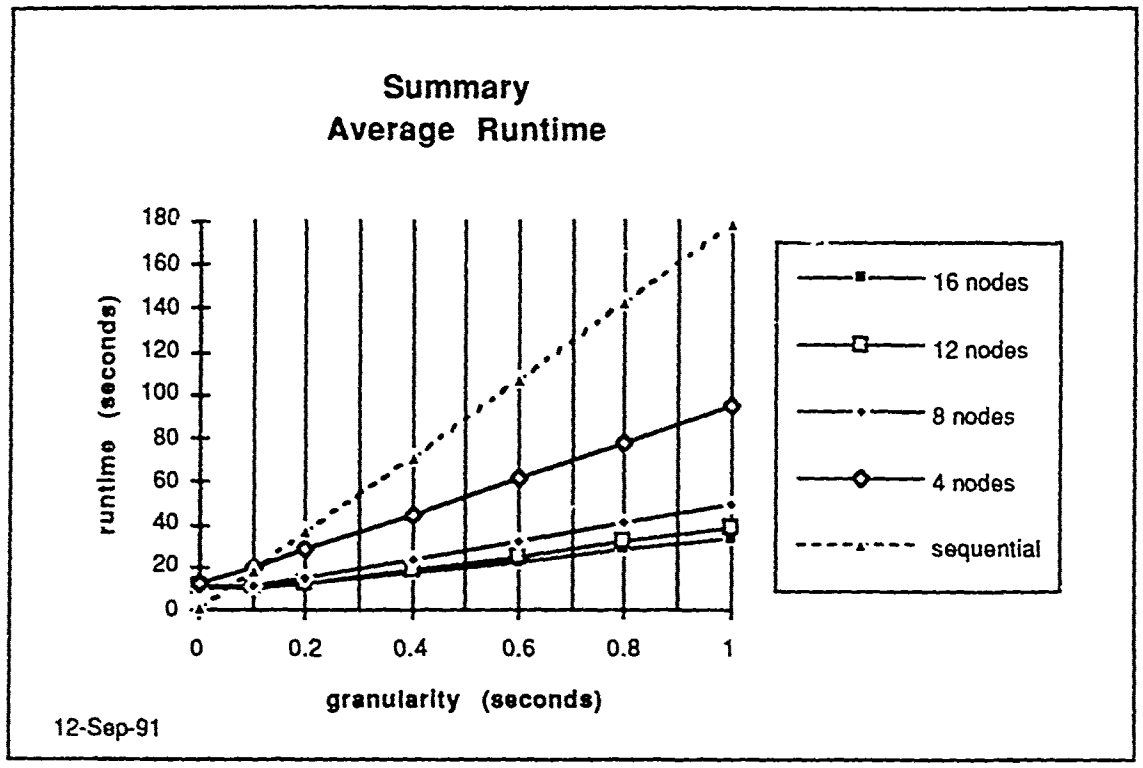


Figure 14. Summary Average Runtime versus Granularity

In the case of the eight, twelve, and sixteen node parallel executions, the zero granularity application test set appears to have runtimes that are approximately the same as the 0.1 second granularity application test set on the average. Since the only difference between the two sets is the granularity of the computations, this phenomenon seemed somewhat surprising until we looked at the average number of rollbacks and the average number of states rolled back for each of the application sets. As figure 15 and 16 show, in the eight, twelve, and sixteen node parallel executions the number of rollbacks and number of states rolled back in the zero granularity case is much higher than at the other granularities tested.

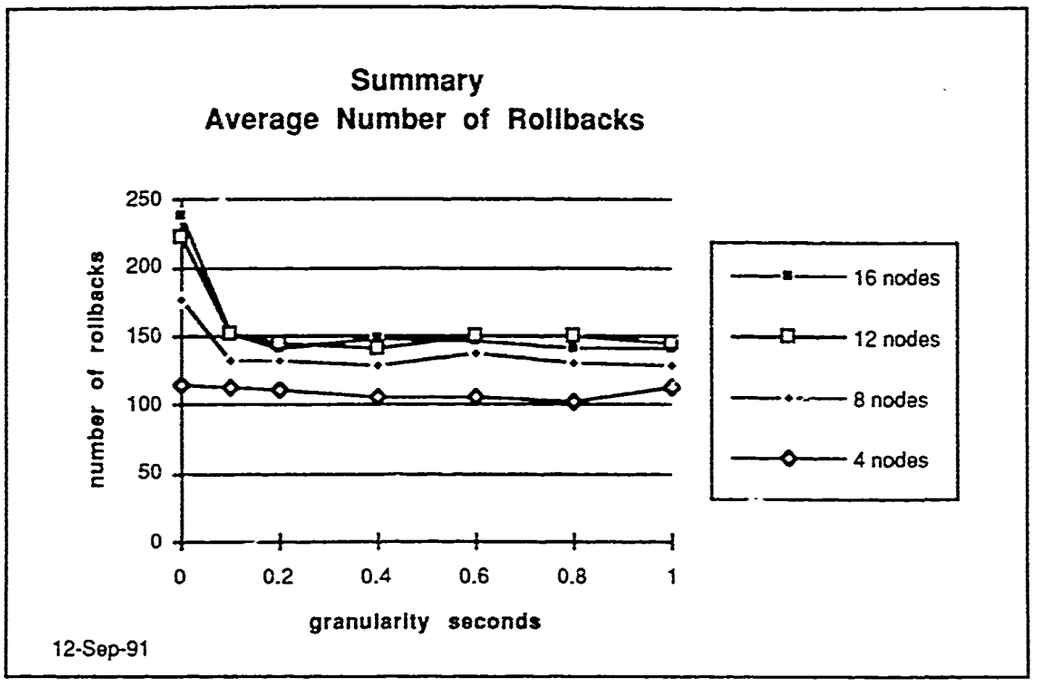


Figure 15. Summary Average Number of Rollbacks versus Granularity

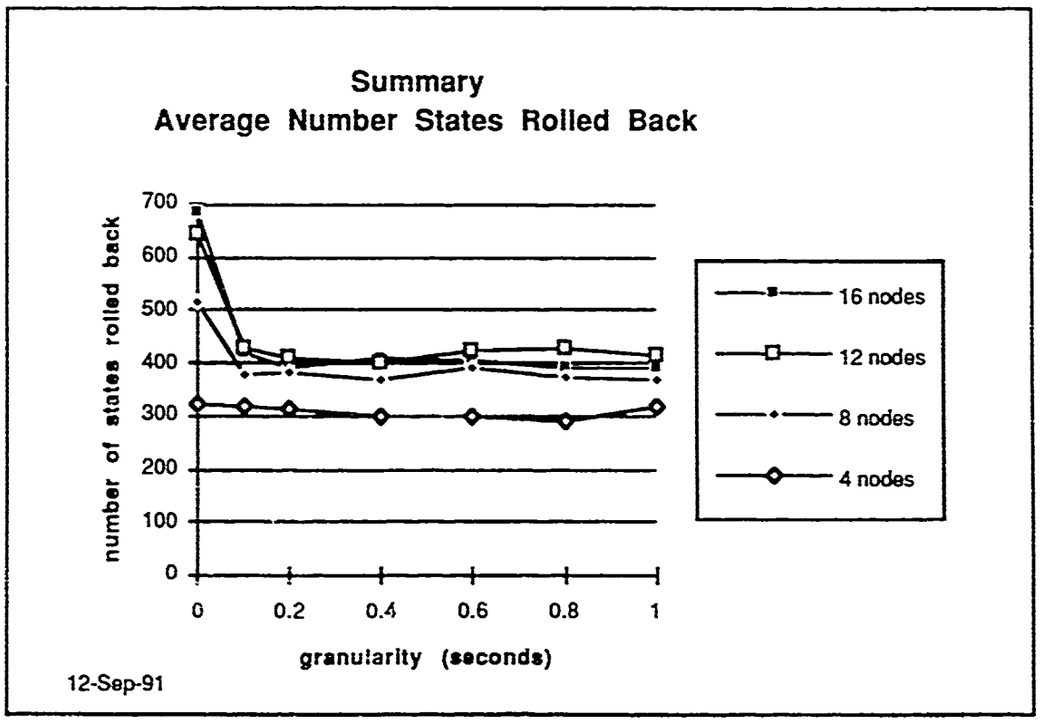


Figure 16. Summary Average Number of States Rolled Back versus Granularity

Although the lower number of rollbacks and states rolled back in the non-zero granularity cases cannot be seen in every individual application tested, the lower number is evident in many of them. Perhaps this phenomenon makes sense if one considers that in the non-zero granularity cases the ratio of computation to communication is higher, which may mitigate the asynchronous nature of the computation to some extent. However, it is interesting that for the non-zero granularity cases, the number of rollbacks and states rolled back varies only slightly.

It is interesting to see how well SaM performs for the large applications in the test set compared to the entire set. Figures 17 through 21 show the same information as figures 8 through 16 but for the ten applications that have the largest number of methods executed. As can be seen in figures 17 and 18, the speedup for the ten largest cases is significantly larger for the non-zero granularities on eight, twelve, and sixteen processors than it was for the test set as a whole. As figures 20 and 21 show, this holds even though the average number of rollbacks and states rolled back is significantly larger for these large cases.

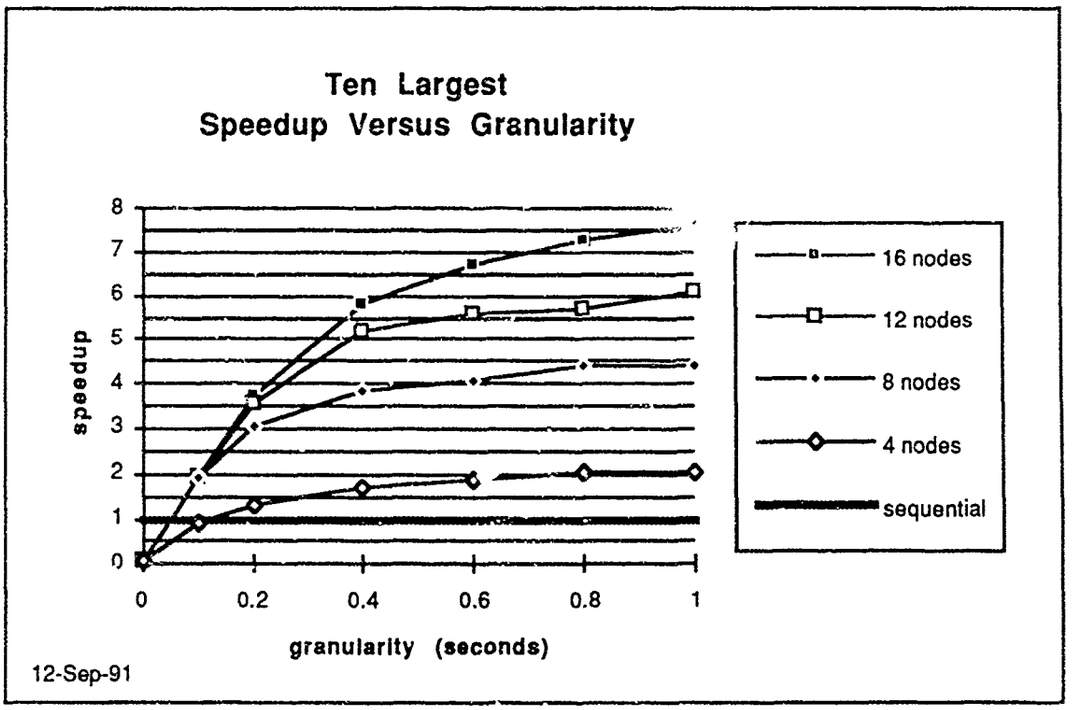


Figure 17. Ten Largest Speedup versus Granularity

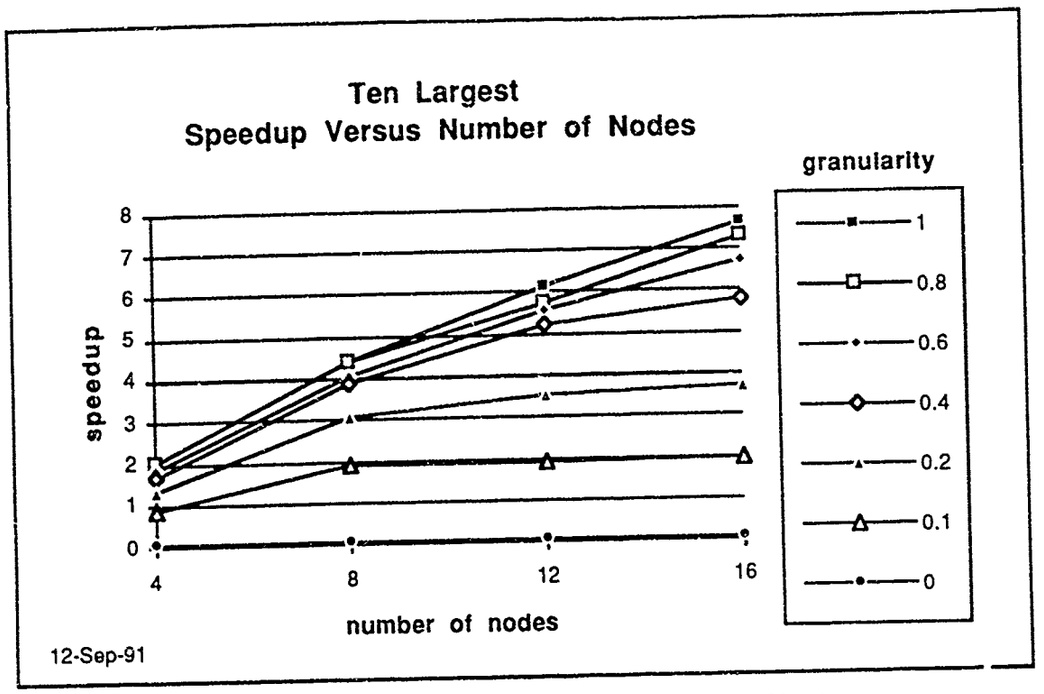


Figure 18. Ten Largest Speedup versus Number of Nodes

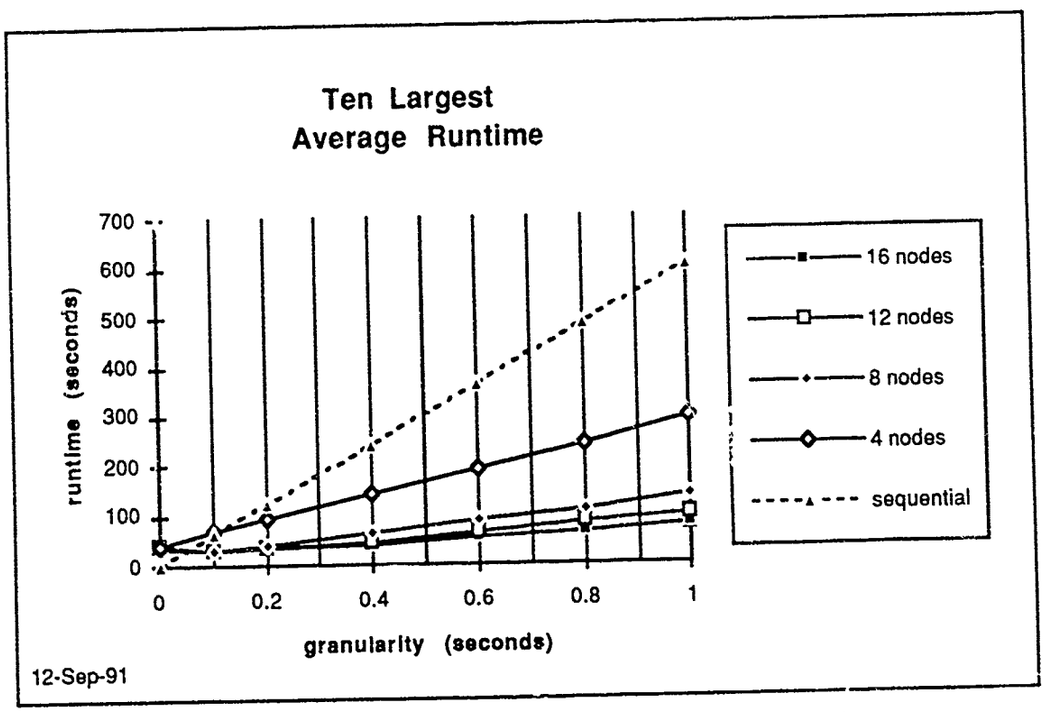


Figure 19. Ten Largest Average Runtime versus Granularity

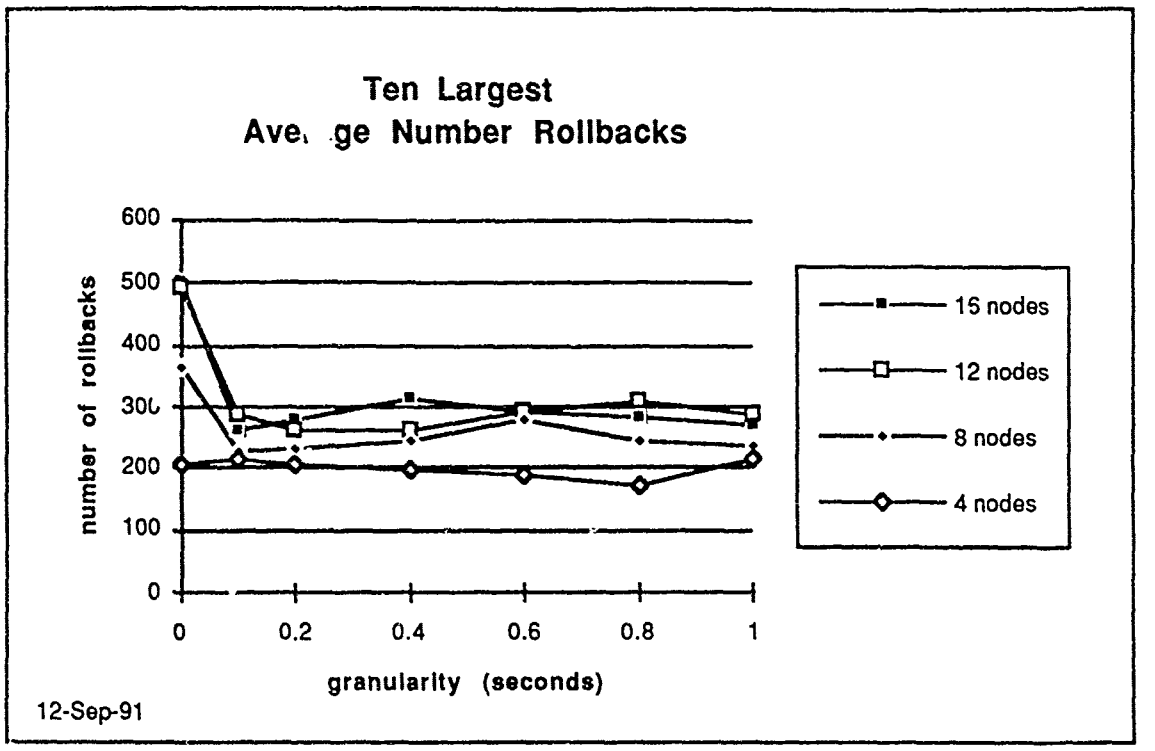


Figure 20. Ten Largest Average Number of Rollbacks versus Granularity

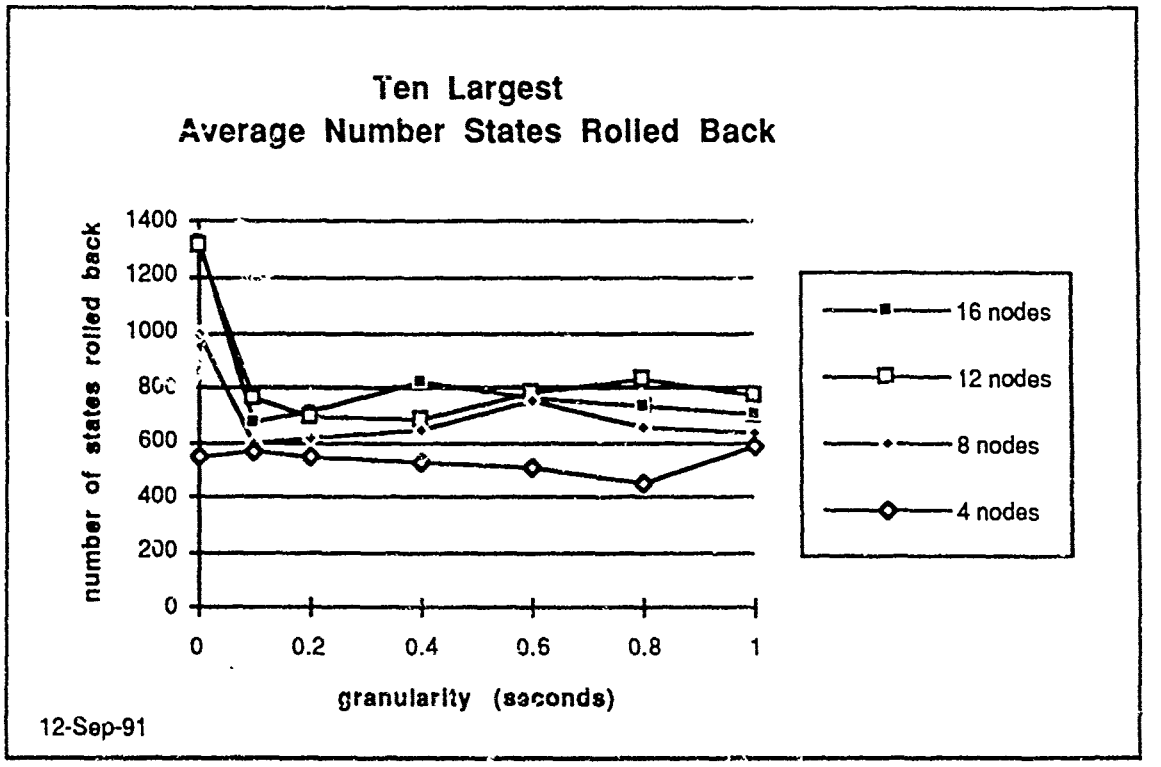


Figure 21. Ten Largest Average Number of States Rolled Back versus Granularity

It is also interesting to see how well SaM performs for the small applications in the test set compared to the entire set. Figures 22 through 26 show the same information as figures 8 through 16 but for the ten applications that have the smallest number of methods executed. As can be seen in figures 22 and 23, the speedup for the ten smallest cases is significantly less for the non-zero granularities on eight, twelve, and sixteen processors than it was for the test set as a whole. As figures 25 and 26 show, this holds even though the average number of rollbacks and states rolled back is significantly less for these small cases.

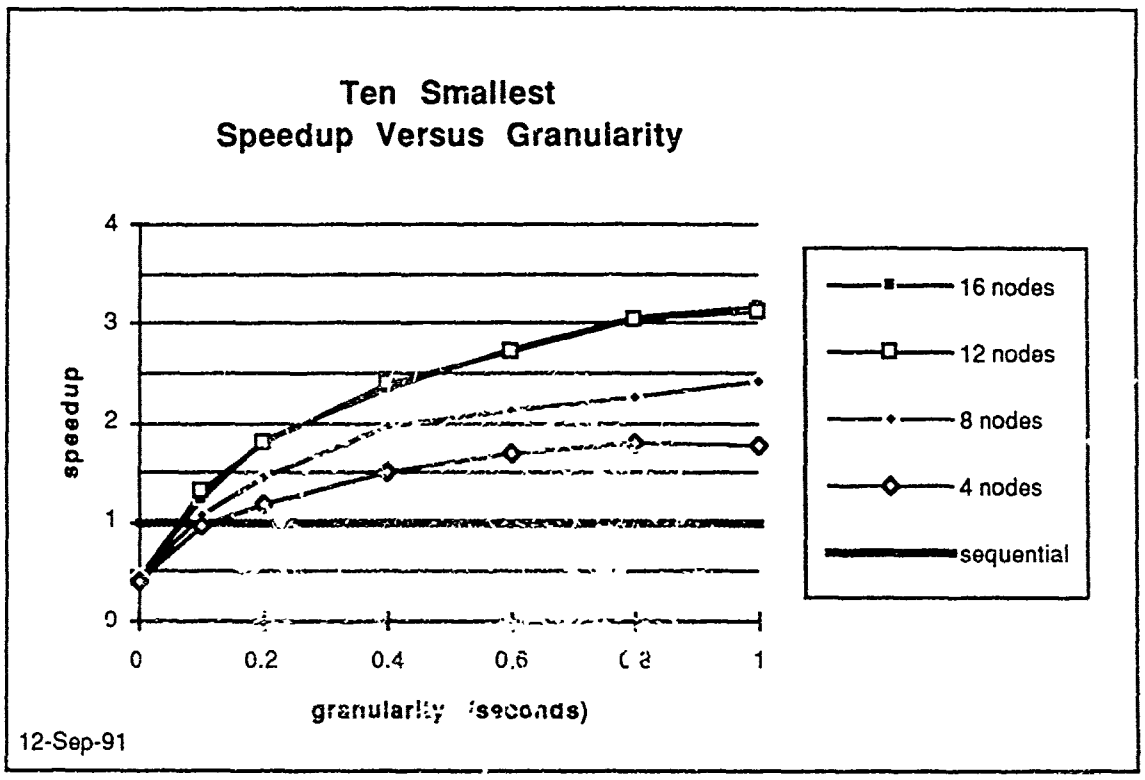


Figure 22. Ten Smallest Speedup versus Granularity

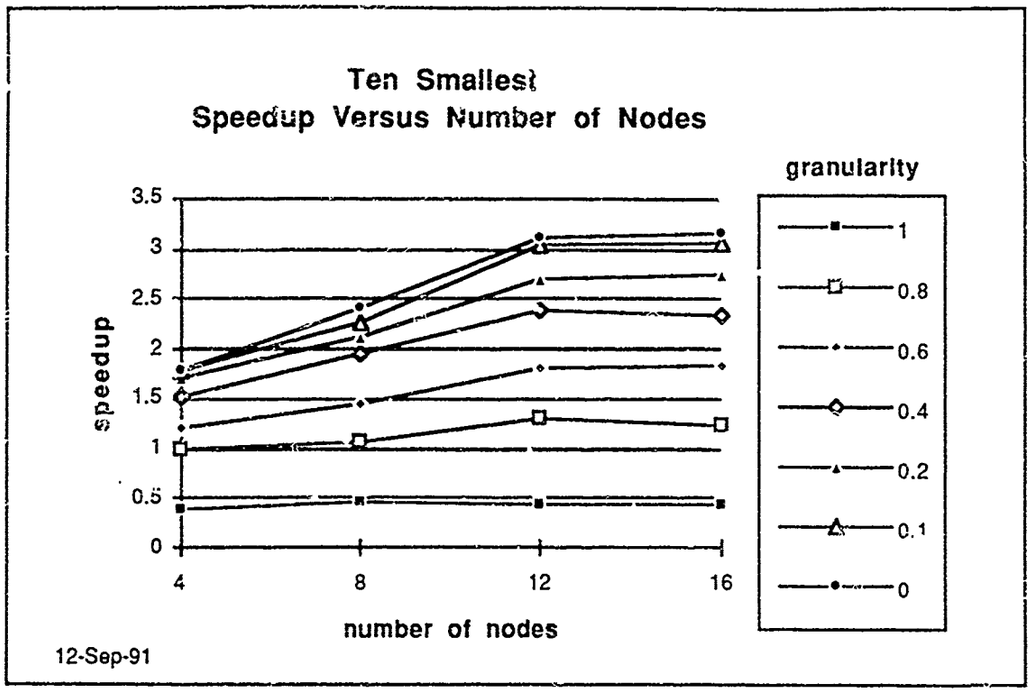


Figure 23. Ten Smallest Speedup versus Number of Nodes

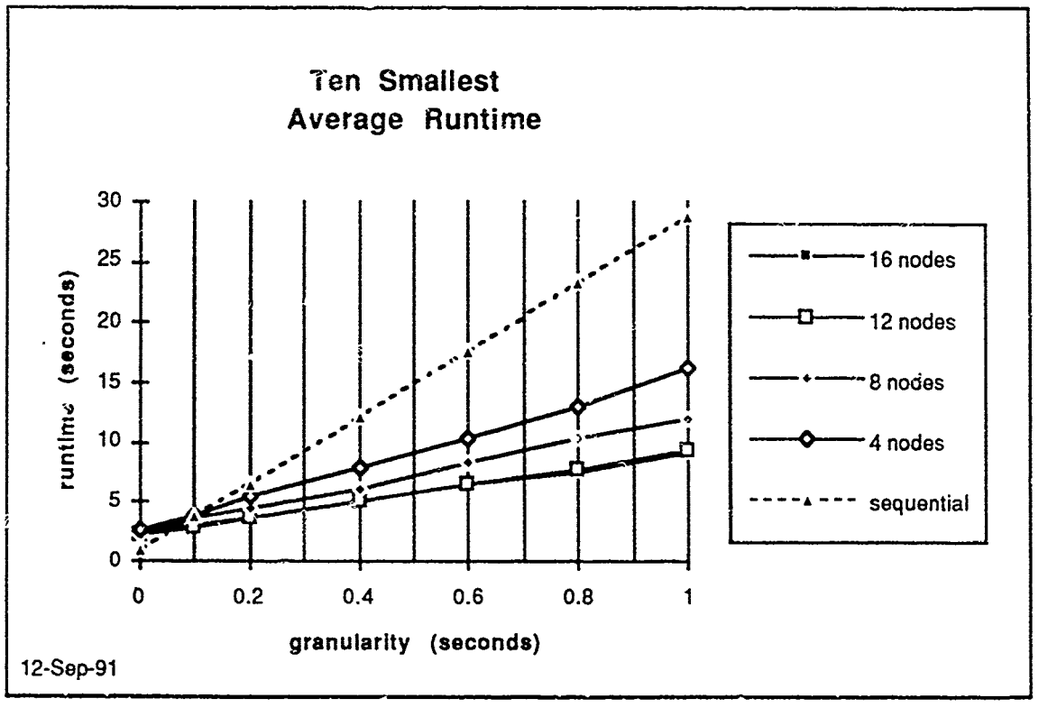


Figure 24. Ten Smallest Average Runtime versus Granularity

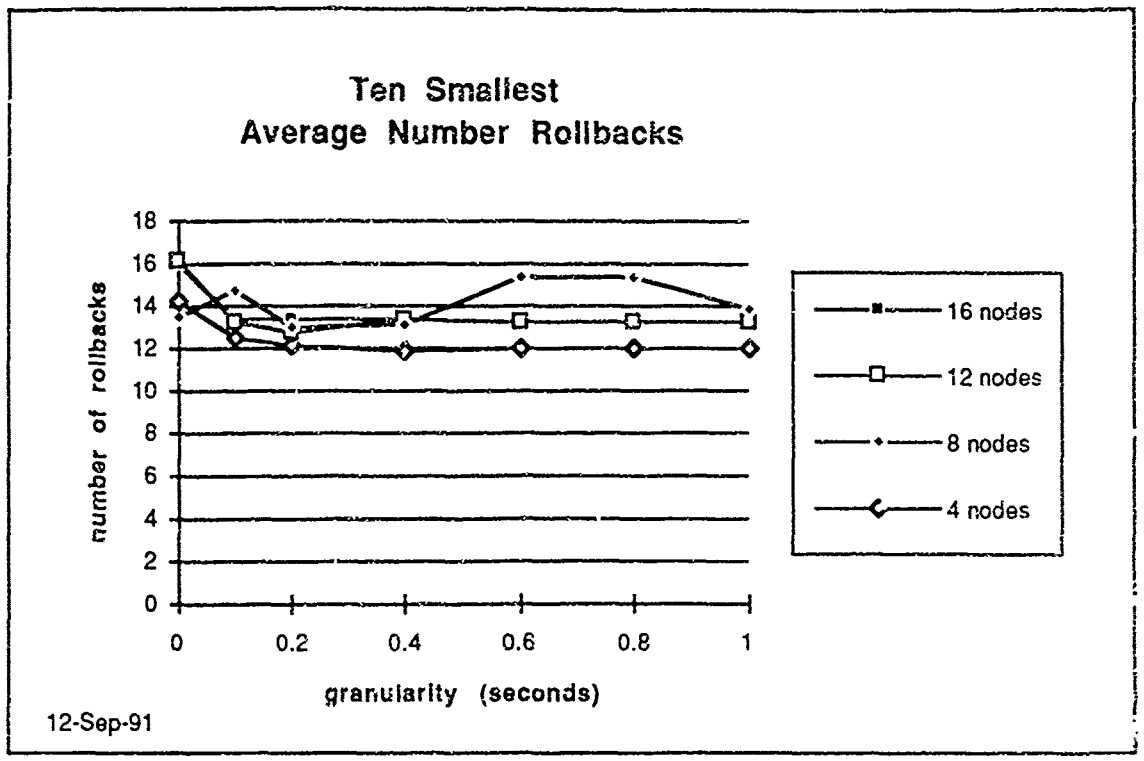


Figure 25. Ten Smallest Average Number of Rollbacks versus Granularity

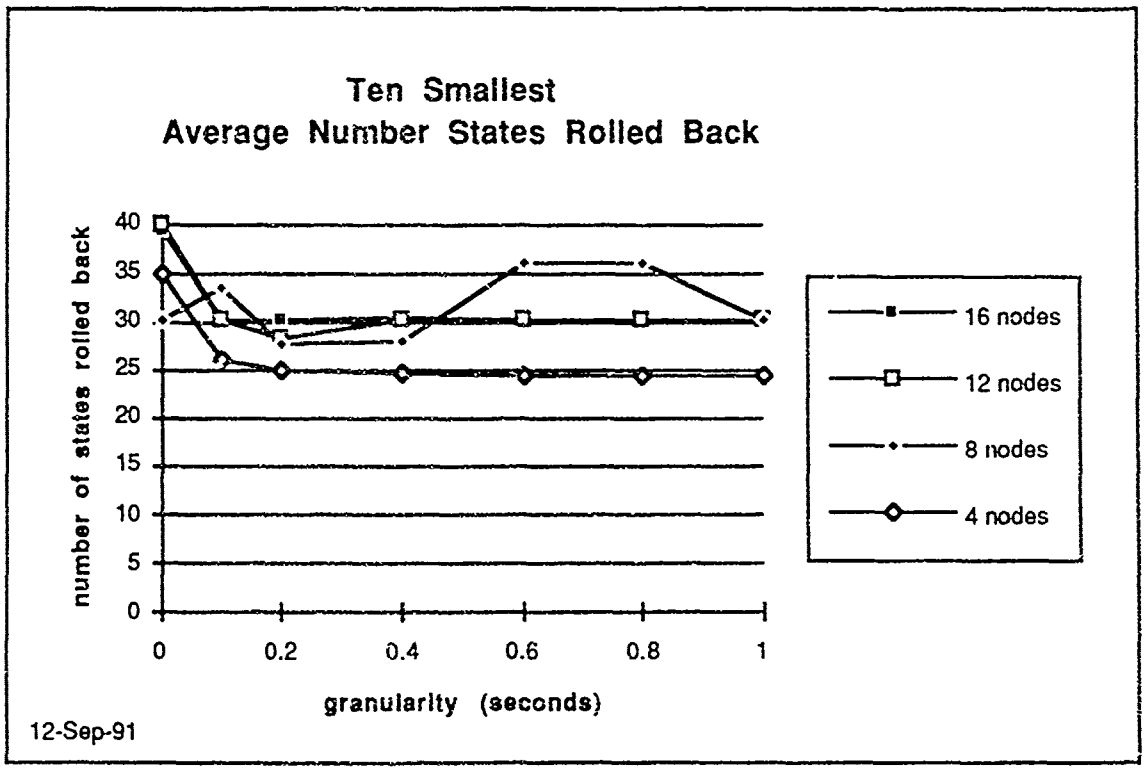


Figure 26. Ten Smallest Average Number of States Rolled Back versus Granularity

SECTION 7

CONCLUSION

SaM assumes little about the underlying architecture except that there is an operating system on each processor that supports typical microprocessor functionality and message passing. Message passing is supported on distributed-memory multicomputers, but it is easy to implement on shared-memory multiprocessors as well. SaM assumes little about an object-oriented application program prior to runtime. The use of resources, both processor and memory, may depend on the input data, for example, the number of messages processed by an object, the number and kind of messages sent by an object, and the number and kind of objects dynamically allocated. SaM assumes the synchronization in the application program may depend on the input data and should be performed with respect to the sequential model, that is, the program should produce the same results as if it were executed on a sequential processor. These assumptions allow application programs to be developed, maintained, extended, and ported more easily. Of course, so much flexibility means that performance of a particular application program on a particular architecture may not be as efficient as it would have been had detailed knowledge of the program's and architecture's characteristics been considered in the application development process.

In this paper, we described the synchronization manager. The synchronization manager uses future objects and checkpoint and rollback to generate and manage synchronization. Context objects manage method execution, including handling recursive cycles of messages correctly. There are a number of objects which comprise the SaM runtime executive. These objects manage synchronization and communication for the application program.

We have used SaM to run application programs written in CPM on a Symult S2010 multicomputer. To test the correctness and performance of SaM against a truly wide variety of application programs, we developed a synthetic application generator program. We used this program to generate applications that vary in their object referencing and creation characteristics, and in the amount of computation they perform per message processed. We ran each of our 378 synthetic applications sequentially on a single Symult S2010 node without the synchronization manager, and in parallel on four, eight, twelve, and sixteen S2010 nodes using the synchronization manager. The application output of each of these 378 programs produced the same values in the multicomputer executions as the equivalent sequential executions.

The three dimensional graph in figure 27 summarizes our results on a set of 378 synthetic application programs. One dimension of the graph shows speed up of the parallel execution using SaM versus running the application sequentially on a single processor

without SaM. Another dimension shows the granularity of the application programs. Granularity indicates how much computation was performed per message processed by each method executed. Not surprisingly as the granularity increases from zero seconds to one second the speedup of the computation improves. The third dimension shows two things simultaneously. The effect of increasing the number of processors is shown; four, eight, twelve, and sixteen processors were used in our tests. It also shows how the ten largest applications (those that executed the greatest number of methods) performed versus the whole set and versus the ten smallest applications (those that executed the fewest number of methods) in the set. As can be seen in figure 27, the speedup for the ten largest cases is significantly larger for the non-zero granularities on eight, twelve, and sixteen processors than it was for the test set as a whole. This holds even though the average number of rollbacks and states rolled back is significantly larger for the large cases.

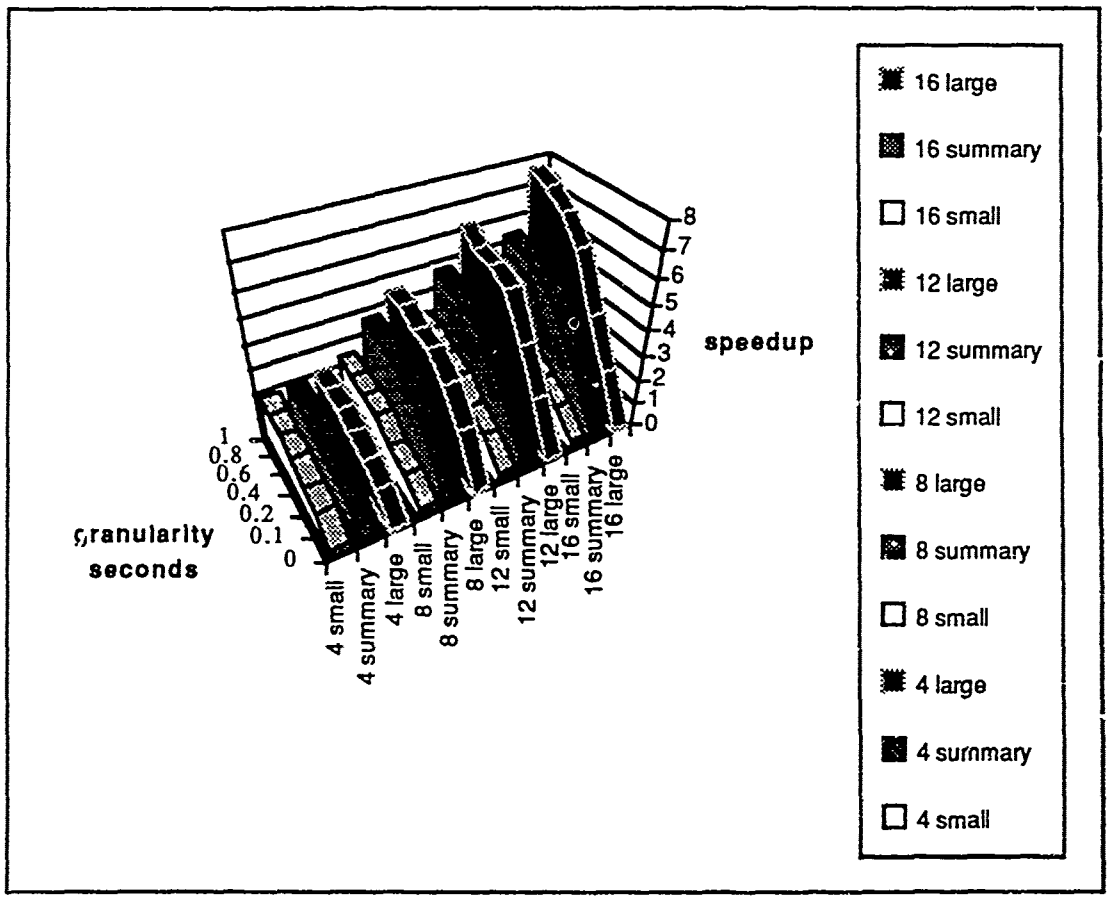


Figure 27. Speedup versus Granularity versus Methods/Processors

In previous work, we investigated extensions of the synchronization manager model of execution to support continued application processing in the presence of hardware failure [Bensley:88] and concurrent language constructs [Prelle:91]. In addition to testing the correctness of the SaM implementation, we have also shown the basic correctness of the

synchronization model mathematically. [Bridgland:91] contains the details of the mathematical proofs, [Prelle:90] contains a more intuitive, descriptive exposition of the arguments and the results. In essence, we have shown that the techniques we developed can automatically manage synchronization. Further, our performance results suggest the viability of this approach.

LIST OF REFERENCES

[Agha:86] Agha, G. A., *ACTORS: A Model of Concurrent Computation in Distributed Systems*, Cambridge, MA: MIT Press, 1986.

[Bensley:88] Bensley, E. H., T. J. Brando, and M. J. Prelle, "An Execution Model For Distributed Object-Oriented Computation," *Proc. Third Annual Conf. on Object Oriented Programming Systems, Languages, and Applications*, San Diego, CA, September 1988.

[Bridgland:91] Bridgland, M. F., J. I. Leivent, and R. J. Watro, *Mathematical Foundations for Time Warp Systems*, MTR 10959, The MITRE Corporation, January 1991.

[Chatterjee:89] Chatterjee, A., "Futures: A Mechanism For Concurrency Among Objects," *Proc. Supercomputing Conference*, Reno, NV, November 1989.

[Chow:87] Chow, E., H. Madan, and J. Peterson, "A Real-Time Adaptive Message Routing Network for the Hypercube Computer," *Proc. Eighth Real-Time Systems Symposium*, IEEE Computer Society, December 1987.

[Halstead:85] Halstead, R. H., "MultiLisp: A Language for Concurrent Symbolic Computation," *ACM Trans. on Prog. Languages and Systems*, pp. 501–538, October 1985.

[Jefferson:87] Jefferson, D., et al., "Distributed Simulation and the Time Warp Operating System," *Proc. Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, November 1987.

[Prelle:90] Prelle, M. J., T. J. Brando, E. H. Bensley, J. I. Leivent, R. J. Watro, and A. M. Wollrath, *Distributed Object-Oriented Programming FY90 Final Report*, MTR 11058, The MITRE Corporation, December 1990.

[Prelle:91] Prelle, M. J., A. M. Wollrath, T. J. Brando, E. H. Bensley, *The Impact of Selected Concurrent Language Constructs on the SaM Runtime System*, OOPS Messenger, ACM Press, Vol. 2, No. 2, April, 1991.

[Reiher:90] Reiher, P., R. Fujimoto, S. Bellenot, and D. Jefferson, "Cancellation Strategies in Optimistic Execution Systems," *Proc. of the SCS Multiconference on Distributed Simulation*, San Diego, CA, January 1990.

[Samadi:85] Samadi, B., "Distributed Simulation, Algorithms and Performance," Ph.D. dissertation, UCLA, 1985.

[Tinker:88] Tinker, P., and M. Katz, "Parallel Execution of Sequential Scheme with ParaTran," *Proc. of the Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988.

APPENDIX

AN OVERVIEW OF THE C+- (CPM) APPLICATION LANGUAGE

The application language is basically a simplified version of C++ which disallows any pointer reference. Thus, the following built-in C operators are disabled, `*`, `&`, and `->`. Although referencing an element of an array through the bracket notation `[]` performs implicit pointer operations, arrays are allowed in the CPM language and are translated into "safe" code by the CPM translator. CPM does not allow dynamic memory allocation at this time. Due to complications dynamic allocation would introduce, we have decided not to support this feature yet, however it may be handled in the future.

In light of the restrictions placed on the application language, we named the language C+- which stands for C plus objects, minus pointers (or CPM as will be referred to in this section).

One cosmetic difference between CPM and C++ is that CPM does not use ANSI standard parameter declarations, whereas C++ does. Instead, the style mimics that found in the first edition of the Kernighan and Ritchie C book which lists the parameters and their declarations separately. For example, compare the following:

```
max(int a, int b)        // C++ declares them within the
{ ...                    // parameter list
```

and,

```
max(a, b)                // CPM declares parameters separately
int a, b;
{ ...
```

Another difference between CPM and C++ is the use of `make_instance` instead of `new` for creating instances.

Format of Appendix

In the *usage* section of each manual page, brackets enclose optional items. If an asterisk appears after the bracket notation, zero or more of the enclosed items are permitted. Keywords and punctuation are in plain text. Any other items in the usage illustration are in italics and should be replaced by code statement(s), type declaration(s), or a user-chosen name—whichever is indicated. Also, throughout the text, CPM keywords appear in a fixed font: for example, `unsigned`.

We assume the reader has some knowledge of both C and C++ syntax and familiarity with object-oriented programming methodologies as well.

Some Basic Rules of CPM

1. All classes must be defined BEFORE their methods are defined. NOTE: there are some exceptions, but this is the safest policy to follow.

2. Local class definitions must appear in the source file BEFORE any variable (instance variable or otherwise) is declared to be a type of that local class. NOTE: as a result, there can be no recursive *local* class definitions.

3. Local class method definitions must appear in the source file BEFORE the method is called from any other method.

4. Increment/decrement operators (++) and (--) are not allowed.

5. The type declarations of all formal parameters for a method must appear in the same order as specified in the parameter list. For example:

```
foo::bar(a, b)
int a;      // correct: the order of type declarations
int b;      //          DOES correspond to formal parameters
{ ...
```

On the other hand,

```
foo::bar(a, b)
int b;      // wrong: the order of type declarations
int a;      //        DOES NOT correspond to formal parameters
{ ...
```

Type Specifiers

In CPM, a type specifier can only be *one* of the following:

```
void, char, short, int, long, float, double, unsigned
```

or,

```
[concurrent] class name
 local       class name
```

Only *one* type specifier is allowed per variable or function declaration. The only exception is the `nfuture` annotation described below.

Examples

```
float f;                // correct
short s, t;             // correct
class clock c;          // correct
local class book b;     // correct
long int i;             // wrong: more than one type specifier
                        //        (long and int)
```

nfuture

Usage

`nfuture` *variable-declaration*

Description

Any valid type specifier can be preceded by the annotation `nfuture` which indicates that the particular variable being declared will never contain a future. If the programmer attempts to store a future value in a variable that is declared not to contain a future, the future will be resolved before the assignment is made, so only a *value* will ever be in an `nfuture` variable; correctness in assignment from futures to non-futures is enforced at the compiler level. For the majority of cases, the `nfuture` declaration *should not* be used before `class` or `concurrent class` declarations, since a variable of type `concurrent class` will contain a future initially when the object is created (via `make_instance`).

Examples

```
nfuture float f;        // correct
nfuture short s, t;     // correct
nfuture class clock c;  // be careful with this one, could cut down on
                        // parallelism
nfuture long int i;     // wrong, more than one type specifier
```

Class Definitions

Concurrent Classes

Concurrent classes in CPM are distributable synchronization managed objects. An instance of a concurrent class should be treated by the application programmer as a pointer to an object of that class. Instance variables of concurrent objects cannot be accessed directly; they may only be accessed via a message sent to the object. Every concurrent object has a unique global object address that can be passed as an argument to any method; that is, a pass by *reference*. Therefore, another object can communicate with this same object by using the object address. In this way, two objects can point to the same concurrent instance.

An instance of a concurrent class must be created via a call to the CPM internal function `make_instance`. CPM provides a default `make_instance` method for each concurrent class, but the application programmer may redefine the `make_instance` method if desired. There are some restrictions on `make_instance` definitions: 1) no message can be sent to any concurrent object within `make_instance`, 2) no future may be resolved within `make_instance`. The determination of messages being sent to concurrent objects (or resolving futures) within the `make_instance` method may be difficult (e.g., if you call a function), so the programmer must be *very* careful when defining this method. One safe policy to follow in order to prevent future resolution is to declare all formal parameters to `make_instance` with the `nfuture` annotation; this will avoid any futures being resolved within the context of `make_instance`.

Local Classes

Local classes are not synchronization managed objects. Therefore, a local class is only accessible to the concurrent object in which it is contained. Unlike concurrent objects, instance variables of local objects are directly accessible (through the dot notation) in the context of the concurrent object. In addition, local objects that are passed as an argument to a method (or returned from a method) are passed by *value* only (i.e., are copied). Two different objects cannot point to the exact same local object.

In CPM, a keyword is added before class definitions (and variable declarations of type `class`) to distinguish between local and concurrent classes. The keywords are *local* and *concurrent* respectively. If no keyword is specified before the class definition, CPM defaults to concurrent class.

concurrent class

Usage

[concurrent] class *name* [is *superclass-name* [, *superclass-name*]*] {
    [*instance-variable-declaration(s)*]
    [*method-declarations(s)*]
};

Description

Generally, a class definition contains instance variable type declarations and method type declarations. CPM supports multiple inheritance which is obtained by using the "is" keyword after the class name, followed by a list of superclasses. Note that similar to a structure definition in C, a class definition must be terminated with a semicolon.

Examples

```
concurrent class clock {            // concurrent class definition
      // instance variables
      int hours, mins, secs;

      // method declarations
      int  get_hours();
      int  get_mins();
      int  get_secs();
      void tick();
};

class cuckoo_clock is clock { // assumes "clock" is a concurrent class
      // instance variables
      int chimes;
      // method
      void cuckoo();
};

concurrent class clock c;   // variable declaration of type concurrent
                            // class
class clock c;              // this declaration is same as the one above

my_hours = c->get_hours();  // concurrent method invocation
```

local class

Usage

```
local class name [ is superclass-name [, superclass-name ]* ] {
        [instance-variable-declaration(s)]
        [method-declarations(s)]
};
```

Description

The definition of a local class is similar to the concurrent class definition illustrated above with the exception of the keyword `local`.

The programmer need not make explicit calls to `make_instance` for local objects because the memory for local objects is allocated when they are declared (whether it be within a concurrent or local object definition, or as a local variable).

Note below that instance variables may be accessed using the dot notation similar to structure member reference in C or member reference in C++.

Examples

```
local class birth_date {        // local class definition
      // instance variables
      int day, month, year;
      // methods
      void print();
      int age();
};

local class birth_date dob;     // variable declaration
my_day = dob.day;               // instance variable reference
my_age = dob.age');             // local method invocation
```

make_instance

Usage
`make_instance` (*class-name* [, *argument*]*)

Description
The `make_instance` function allocates space for a concurrent object. For a more complete description of `make_instance`, see the above description about concurrent classes. How to redefine `make_instance` for a particular concurrent class will be covered in the next section which explains the syntax of method definitions.

Examples
```
class clock plain_clock;
class cuckoo_clock fancy_clock;

// making use of the default make_instance for the class clock...
plain_clock = make_instance(clock);

// if the application programmer redefines make_instance to accept
// one or more arguments, the following is possible...
fancy_clock = make_instance(cuckoo_clock, NumChimes);
```

Method Definitions

This section explains method invocation, method definition, and how to redefine the default `make_instance` definition for a concurrent object.

`method invocation (message sending)`

Usage
object`.`*method-name* ([*parameter* [, *parameter*] *])
object`->`*method-name* ([*parameter* [, *parameter*] *])

Description
Method invocation (or sending a message to an object) has the same syntax as in C++. The `self` keyword is provided and is similar to the `this` keyword in C++. The use of `self` differs slightly from `this`, in that `self` must be specified as the target object if a method will be invoked on the object which the current method is acting on. In C++, if a method call appears without a destination object specified, `self` is assumed.

Examples
In order to send a message to a local object, the following syntax is appropriate:

```
x.print();          // local object method invocation
self.help();        // send the help message to myself
```

as opposed to the following syntax for concurrent objects:

```
x->print();         // concurrent object method invocation
self->help();       // send the help message to myself
```

method definition

Usage

return-type class-name::*method-name* ([*parameter* [, *parameter*] *])
    parameter-declaration(s)
{
    statement(s)
}

Description

The syntax for method definition is quite similar to C++ and does not differ between local and concurrent object method definitions. Within the body of a method, instance variables of the class (for which the method is being defined) can be referenced by their name. A method of the same class must be invoked using the keyword `self` as the object name. For an example, see the usage of `self` in the example below.

Examples

```
// here is the print method for the local class birth_date
void birth_date::print()
{
      printf("Your date of birth is: %2d/%2d/%2d\n",
            month, day, year);
}

int birth_date::age(currentDay, currentMonth, currentYear)
{
      int age;

      // calculate age
      age = currentYear - year;
      if (currentMonth == month) {
            if (currentDay == day)
                  printf("Happy Birthday!\n");
            else if (currentDay < day)
                  age = age - 1;
      } else if (currentMonth < month)
            age = age - 1;

      // print date of birth
      self.print();

      return age;
}
```

make_instance method definition

Usage

```
class class-name class-name::make_instance ( [ parameter [, parameter ] * ] )
        parameter-declaration(s)
{
        statement(s)
}
```

Description

The caveats for `make_instance` are explained above in the section which describes concurrent classes. Briefly, no method invocations or future resolutions may be permitted within `make_instance`.

Examples

```
class cuckoo_clock cuckoo_clock::make_instance(the_chimes)
{
        chimes = the_chimes;
}
```

declaring instances

Usage

`local class` class-*name instance-name* [, *instance-name*] ;
[`concurrent`] `class` class-*name instance-name* [, *instance-name*] ;
[`concurrent`] `class` class-*name* `?` *instance-name* [, *instance-name*] ;

Description

Two different types of declarations are provided to declare variables (or functions, arrays, etc.) that contain (or return) instances of classes. The first two, using `class` *class-name* as a type specifier, declare variables that contain instances of the specific named class only. The third, using `class` *class-name* `?` as a type specifier, declares variables that contain instances of the named class or one of its subclasses. This second type of declaration provides the semantics of polymorphism. No polymorphism is allowed in local classes.

Examples

```
class clock? c1, c2;     // c1 and c2 can either be an instance of
                         // clock or one of its subclasses

// both of the following are valid...
c1 = make_instance(clock);
c2 = make_instance(cuckoo_clock, NumChimes);
```

Preprocessor Directives

Similar to the C-preprocessor, the CPM translator handles a few preprocessor directives as well. However, these directives are prefixed with percent (%) instead of with the pound sign (#) as in C. Also similar to the "#" in a C-preprocessor directive, the "%" in a CPM-preprocessor directive must appear in the first column of the line on which it appears.

%cpm_block ... %cpm_end

Usage

```
%cpm_block
statement(s)
%cpm_end
```

Description

The "`%cpm_block ... %cpm_end`" preprocessor directive should be wrapped around a block of code that the POOPC translator should not parse (and, therefore, will not appear in the output), but the CPM translator should parse. The `%cpm_block` directive is mainly used to distinguish code which will be used only in the parallel version of a user application (in the version of the application running with the synchronization manager).

Examples

```
%cpm_block
printf("This will only be printed in the cpm version:\n");
%cpm_end
```

See Also

`%poopc_block ... %poopc_end`

%declare

Usage
`%declare` *variable-or-function-declaration*

Description
The `%declare` preprocessor directive makes the CPM translator aware of the variable or function declaration, but the declaration does not appear in the output of the translator.

Internal C functions referenced in the user application (any functions not appearing in the file being preprocessed by CPM), must be either declared explicitly using a valid type specifier recognized by CPM, or must be declared using the `%declare` preprocessor directive.

Examples
```
%declare void printf();
%declare void fprintf();
%declare int fopen();
```

See Also
`%declare_block ... %declare_end`

%declare_block ... %declare_end

Usage
```
%declare_block
variable-or-function-declaration(s)
%declare_end
```

Description
The "`%declare_block ... %declare_end`" preprocessor directive performs the same basic function as the `%declare` preprocessor directive, but encloses a block of code instead of prefixing only a single line of code.

Examples
```
%declare_block
      void printf();
      void fprintf();
      int fopen();
%declare_end
```

See Also
`%declare`

%ignore_block ... %ignore_end

Usage
```
%ignore_block
statement(s)
%ignore_end
```

Description
The "`%ignore_block ... %ignore_end`" preprocessor directive performs the same basic function as the `%pass` preprocessor directive, but encloses a block of code instead of prefixing only a single line of code. The block of code will be passed through either POOPC or CPM untouched.

Examples
```
%ignore_block
#include <stdio.h>
FILE *infile;
%ignore_end
```

See Also
`%pass`

%pass

Usage
`%pass` *statement-or-declaration*

Description
The `%pass` preprocessor directive allows a line of code to be ignored by the CPM translator, and the line appears exactly "as is" in the output of the translator.

Examples
```
%pass #include <stdio.h>
%pass FILE *infile;
```

See Also
`%ignore_block ... %ignore_end`

%poopc_block ... %poopc_end

Usage

```
%poopc_block
statement(s)
%poopc_end
```

Description

The "%poopc_block ... %poopc_end" preprocessor directive should be wrapped around a block of code that the CPM translator should not parse (and, therefore, will not appear in the output), but the POOPC translator should parse. The %poopc_block directive is mainly used to distinguish code which will be used only in the sequential version of a user application (not in the parallel version of the application running with the synchronization manager).

Examples

```
%poopc_block
%pass #include <stdio.h>
%poopc_end

%poopc_block
/* only the seqential version of the program will have a main */
main()
{
      /* ... */
}
%poopc_end
```

See Also

%cpm_block ... %cpm_end

Local Class Library

A library of local classes has been implemented for the convenience of the application programmer. These classes are parameterized by type; thus, by using preprocessor symbolic constants, the programmer can specify the data type of the element contained in the data structure represented by the class (stack, queue, or linked list). Consequently, it is very simple to create a stack of integers, characters, strings, or a class of any kind. Likewise, queues and linked lists of integers, characters, strings, etc., can be constructed.

For the local classes stack, queue, and list, there are several symbolic constants that should be defined to specify the user-selected name of the class, the size of the structure (stack, queue, or list), the data type of the elements in the structure, and how to print out an element (in the stack, queue, or list). The section, *Symbolic Constants*, in each library class description lists the names of the symbolic constants that can be defined if the user wishes. Each of these constants have some default value. For example, the element type for each local class defaults to `nfuture int`. Default values for other symbolic constants will be mentioned in their respective sections below.

The procedure for defining a local class (using this library) with a user-specified element type is as follows:

```
#define symbolic-constant1  value1
#define symbolic-constant2  value2
...
#include local-class-header-file
#undef symbolic-constant1
#undef symbolic-constant2
...
```

For example, if the user wanted to use two queues, one with elements of type `local class foo` and the other with elements of type `local class bar`, the following code would be appropriate:

```
#define QueueName        FooQueue
#define QueueElement     local class foo
#define QueueSize        100
#define PrintQueueElement(e)  e.print()
#include "/vb/ann/cpm/lib/queue_lib.h"
#undef QueueName
#undef QueueElement
// do not need to #undef QueueSize or PrintQueueElement since
// their value does not change for the next queue definition.
#define QueueName        BarQueue
#define QueueElement     local class bar
#include "/vb/ann/cpm/lib/queue_lib.h"
#undef QueueName
#undef QueueElement
#undef QueueSize
#undef PrintQueueElement
...
```

The programmer now has access to all methods defined for the generic queue. Local instances of FooQueue and BarQueue are used as follows:

```
local class FooQueue f;
local class BarQueue b;

f.init();
b.init();
...
```

The following manual pages describe the local class definitions that are contained in the library: string, stack, queue, and list. Other local classes will be added to the library as necessary.

`string`

Usage

```
#include "/vb/ann/cpm/lib/string_lib.h"
smString identifier;
medString identifier;
lgString identifier;
hugeString identifier;
```

Methods

```
void init(char *cp);      // initialize string
void copy(stringType s);  // copy from string argument
void set(int i, char c); // set the i-th element to the character
nfuture char get(int i); // get the i-th element in the string
void print();             // print out the string
```

Description

There are four classes that implement strings: `smString`, `medString`, `lgString`, and `hugeString` (respectively small, medium, large, and huge). A small string can be up to 20 characters in length; a medium string can be up to 40 characters long; a large string can be up to 80 characters long; finally, a huge string can be up to 256 characters in length. These character strings are treated as null terminated strings. Subscripting of arrays is zero-based.

NOTE: Be careful to make sure that when passing an argument of any string type that the type matches the formal parameter of the method. For example, if a method declares its parameter as a medString, only a medString should be passed as an argument to the method. If an smString is passed instead, an error will occur. The compiler will not detect these types of errors, therefore, the programmer should be very careful to use the appropriate string types.

The `init()` method is used to initialize a string to a particular character string constant (a character string between quotes). The use of a character pointer in the `init()` method is an exception in the CPM language (special hand coding was necessary to implement a pointer—pointers are not generally available in CPM).

The `copy()` method takes another string (of the *same* type) as an argument and copies the contents of that string to itself.

The `set()` method sets the i-th element in the string to the character c.

`get()` returns the i-th character in the string.

The `print()` method prints out the string.

The length of the string can be retrieved by accessing the `length` member of the string class.

Examples

```
#include "/vb/ann/cpm/lib/string_lib.h"
...
smString s, t;
s.init("hello world!");
t.copy(s);
s.print();
printf("\n");
printf("The fifth character in the string 's' is: %c\n", s.get(4));
s.set(0, 'y');
s.print();
printf("\n");
t.print();
printf("\n");
printf("The string 't' is %d characters long.\n", t.length);

=> hello world!
=> The fifth character in the string 's' is: o
=> yello world!
=> hello world!
=> The string 't' is 12 characters long.
```

Usage

```
#include "/vb/ann/cpm/lib/stack_lib.h"
```

Methods

```
void init()                    // initialize stack
void push(StackElement e)      // push an element on the stack
StackElement pop()             // pop an element from the stack
StackElement top()             // get the top element from the stack
nfuture int size()             // get the size of the stack
nfuture int isEmpty()          // test if stack is empty
void print()                   // print out stack contents
```

Symbolic Constants

```
StackName                      // name of the stack
StackSize                      // stack size
StackElement                   // data type of elements in stack
PrintStackElement(e)           // macro for printing a stack element
```

Defaults

```
#define StackName              stack
#define StackSize              50
#define StackElement           nfuture int
#define PrintStackElement(e)   printf("%d", e)
```

Description

The local class stack implements a LIFO stack. A stack should be initialized before using the `init()` method.

The `push()` method adds an element to the top of the stack.

The `pop()` method deletes the element from the top of the stack and returns that element.

The `top()` method returns the first element on the stack (the top element).

`size()` returns the number of elements in the stack.

`isEmpty()` returns `TRUE` if the stack is empty, otherwise it returns `FALSE`.

The `print()` method prints out the contents of the stack.

Examples

```
#define StackName              CharStack
#define StackElement           nfuture char
#define StackSize              100
#define PrintStackElement(e)   printf("%c", e)
#include "/vb/ann/cpm/lib/stack_lib.h"
#undef  StackName
#undef  StackElement
#undef  StackSize
#undef  PrintStackElement
...
local class CharStack cStack;
int i;

cStack.init();
cStack.print();
for (i=0; i<4; i=i+1)
      cStack.push('a' + i);
printf("The size of the stack 'cStack' is: %d\n", cStack.size());
cStack.print();

=> (empty)
=> The size of the stack 'cStack' is: 4
=> a
=> b
=> c
=> d
```

Usage

```
#include "/vb/ann/cpm/lib/queue_lib.h"
```

Methods

```
void init()                    // initialize queue
void add(QueueElement e)       // add element to queue
QueueElement delete()          // delete the first element from queue
QueueElement first()           // get first element from queue
QueueElement last()            // get last element from queue
nfuture int size()             // get the size of the queue
nfuture int isEmpty()          // test if queue is empty
void print()                   // print out queue contents
```

Symbolic Constants

```
QueueName                      // name of the queue
QueueSize                      // queue size
QueueElement                   // data type of elements in the queue
PrintQueueElement(e)           // macro for printing a queue element
```

Defaults

```
#define QueueName               queue
#define QueueSize               50
#define QueueElement            nfuture int
#define PrintQueueElement(e)    printf("%d", e)
```

Description

The local class queue implements a FIFO queue. Before using a queue, it must be initialized using the `init()` method.

The `add()` method, appends the element to the end of the queue.

The `delete()` method, removes the first element on the queue and returns that element.

The `first()` method returns the first element on the queue; similarly, `last()` returns the last element in the queue.

`size()` returns the number of elements in the queue.

`isEmpty()` returns `TRUE` if the queue is empty, otherwise it returns `FALSE`.

The `print()` method prints out the contents of the queue.

Examples

See the example in introductory explanation to this section.

Usage

```
#include "/vb/ann/cpm/lib/list_lib.h"
```

Methods

```
void init()                              // initialize linked list
void insert(ListElement e)               // add element to linked list
nfuture int remove(ListElement e)        // remove element from list
ListElement find(ListElement key, ListElement default)
                                         // find element in list
nfuture int size()                       // get the size of the list
nfuture int isMember(ListElement e) // test element for list membership
nfuture int isEmpty()                    // test if list is empty
void print()                             // print out list contents
```

Symbolic Constants

```
ListName                // name of the list
ListSize                // list size
ListElement             // data type of elements in linked list
PrintListElement(e)     // macro for printing an element in linked list
ListElementEqual(e1,e2) // macro to test element equality (for isMember)
ListElementLess(e1,e2)  // macro to compare elements (for insert)
ListElementKey(e1,e2)   // macro to compare element keys (for find)
```

Defaults

```
#define ListName                    list
#define ListSize                    50
#define ListElemen                  nfuture int
#define PrintListElement(e)         printf("%d", e)
#define ListElementEqual(e1,e2)     (e1 == e2)
#define ListElementLess(e1,e2)      (e1 < e2)
#define ListElementKey(e1,e2)       (e1 == e2)
```

Description

The local class `list` implements a sorted linked list whose sort order is determined by the `ListElementLess` macro. Before using a linked list, it must be initialized using the `init()` method.

The `insert()` method adds the element to the linked list in sorted order.

The `remove()` method deletes the element that is equivalent to the one passed as the argument as determined by the `ListElementEqual` macro. `remove()` returns `TRUE` if the element is deleted from the list, otherwise `FALSE` is returned (the element was not found).

The `find()` method returns the first element that matches the key element, using the `ListElementKey` macro for comparison between the element in the list and the key element passed as an argument. If no element in the linked list matches the key element, the default element is returned instead.

The `isMember()` method returns TRUE if the element is equivalent to a member in the list (using the `ListElementEqual` macro), otherwise it returns FALSE.

`size()` returns the number of elements in the linked list.

`isEmpty()` returns TRUE if the linked list is empty, otherwise it returns FALSE.

The `print()` method prints out the contents of the linked list.

Examples

```
local class employee {
      int idNumber;
      smString lastName;
      smString firstName;
      char mi;
      ...
      void init();
      nfuture int isEqual();
      void print();
};

// assume employee::init(), employee::isEqual(), and employee::print()
// are defined
#define ListName                    EmployeeList
#define ListElement                 nfuture local class employee
#define PrintListElement(e)         e.print()
#define ListElementEqual(e1,e2)     (e1.isEqual(e2))
#define ListElementLess(e1,e2)      (e1.idNumber < e2.idNumber)
#define ListElementKey(e1,e1)       (e1.idNumber == e2.idNumber)
#include "/vb/ann/cpm/lib/list_lib.h"
#undef ListName
#undef ListElement
#undef PrintListElement
#undef ListElementEqual
#undef ListElementLess
...

local class EmployeeList elist;
local class employee emp;
smString last, first, null;
int empNum;

elist.init();                  // initialize employee list
first.init("Jacob");           // initialize first name
last.init("Worker");           // initialize last name
empNum = 2869;                 // employee number
null.init("");                 // initialize null string

emp.init(empNum, first, 'Q', last); // init employee record
elist.insert(emp);             // insert employee record in list
emp.init(empNum, null, '\0', null); // init null record with key
(elist.find(emp)).print();     // lookup employee by key and print

=> 2869 Jacob Q. Worker
```